# Métodos de Ordenação

João Francisco Teles da Silva
*Engenharia da Computação*
*CEFET-MG*
Divinópolis, Brasil
joaoteles0505@gmail.com

Joaquim Cézar Santana da Cruz
*Engenharia da Computação*
*CEFET-MG*
Divinópolis, Brasil
joaquimcezar930@gmail.com

Matheus Emanuel da Silva
*Engenharia da Computação*
*CEFET-MG*
Divinópolis, Brasil
memanuel643@gmail.com

***Abstract*—The analysis and implementation of sorting algorithms are fundamental to computer science, essential for efficient data organization in applications ranging from databases to search tools. Among the notable algorithms, Cocktail Sort and Bucket Sort stand out due to their distinct approaches. Cocktail Sort, a bidirectional variant of Bubble Sort, improves efficiency by sorting elements in both directions in each pass. Bucket Sort, on the other hand, divides elements into multiple 'buckets' and sorts each one individually, making it particularly effective for uniformly distributed datasets. This article delves into the functionalities, advantages, disadvantages, and ideal applications of these algorithms, providing a comprehensive understanding of how the choice of a sorting algorithm impacts execution time and memory usage.**

***Index Terms*—Cocktail Sort, Bucket Sort, Sorting Algorithms, Algorithm Efficiency, Computational Complexity,Computer Science, Execution Time, Memory Usage**

# I Introdução

A análise e implementação de algoritmos de ordenação são fundamentais para a compreensão dos princípios da ciência da computação. Estes algoritmos são cruciais para a organização eficiente de dados, um aspecto vital em diversas aplicações, como sistemas de banco de dados ou ferramentas de busca e classificação. Dentre os numerosos métodos de ordenação descritos na literatura, alguns se destacam por sua aplicabilidade prática e relevância acadêmica. Neste contexto, dois algoritmos de ordenação se destacam por suas abordagens distintas: o Cocktail Sort e o Bucket Sort.

O Cocktail Sort, também conhecido como Shaker Sort/ Bidiretional Bubble sort, é uma variação do algoritmo Bubble Sort. O algoritmo Bubble sort trabalha, de maneira simplificada, colocando os maiores números na parte final do array e os menores números na parte inicial do array. Para fazer isso, a cada execução do laço de repetição, o algorítmo volta a posição inicial do vetor. Diante disso, tendo como exemplo uma ordenação aleatória de números, os menores valores tendem a se movimentar por menos casas, enquanto os números maiores se deslocam por mais casas de acordo com as comparações. Esse cenário faz com que os maiores números se tornem 'Lebres', enquanto os menores números, que se movem menos, se tornem 'tartarugas'.

Desse modo, para que todos os números possam fazer deslocamentos semelhantes, e se tornarem 'lebres', foi criado o Cocktail Sort, que ordena os elementos em ambas as direções a cada passagem pela lista.Essa característica permite uma eficiência ligeiramente melhor em certos casos em comparação com o Bubble Sort tradicional.

Por outro lado, o Bucket Sort adota uma abordagem diferente, dividindo os elementos em vários 'baldes' (buckets) e ordenando cada balde individualmente. Este método é particularmente eficiente para conjuntos de dados que estão distribuídos de maneira uniforme em um intervalo específico.

Ao longo deste artigo, será explorado em detalhes o funcionamento de ambos os algoritmos, suas vantagens e desvantagens, e os cenários ideais para sua aplicação. A análise dos algoritmos Cocktail Sort e o Bucket Sort fornecerá uma compreensão mais profunda sobre como a escolha de determiando algoritmo de ordenação pode influenciar em diferentes aspectos, como o tempo de execução e a memória utilizada.

# II Metodologia

## A. Cocktail Sort

O Cocktail Sort é um algoritmo de ordenação bidirecional, que ordena os elementos em ambas as direções a cada passagem pelo vetor.Inicialmente, o algoritmo percorre o vetor na direção crescente, comparando pares de elementos adjacentes e trocando-os se estiverem fora de ordem. Durante esta primeira passagem, os maiores elementos são gradualmente deslocados para o final da lista.Após a conclusão da passagem inicial, o algoritmo realiza uma segunda varredura no sentido contrário, do final para o início do vetor. Nesta etapa, os elementos são comparados novamente em pares, e os menores valores são movidos progressivamente para o início da lista.

O algoritmo, quando analisada sua complexidade de tempo, possui complexidade de $O(N^2)$ para o seu caso médio e para o seu pior caso. Quando tratado o melhor caso, o Cocktail Sort possui uma característica única que é detectar quando o vetor ou lista já está ordenado de maneira correta. Dessa maneira, a execução é terminada após percorrer uma vez o vetor, tendo assim, complexidade de O(n) em seu melhor caso.

Analisando o gráfico 1 abaixo, fica exposto que, com o aumento de N (tamanho da entrada) o tempo de execução do algoritmo cresce exponencialmente. Em comparação do pior caso para com o melhor, para valores de N pequenos, a diferença entre os tempos de execução é irrisória, sendo de apenas alguns nanosegunos, porém para valores de N maiores, o crescimento exponencial se sobrepõe ao crescimento linear, fazendo com que o tempo de execução seja muito maior.

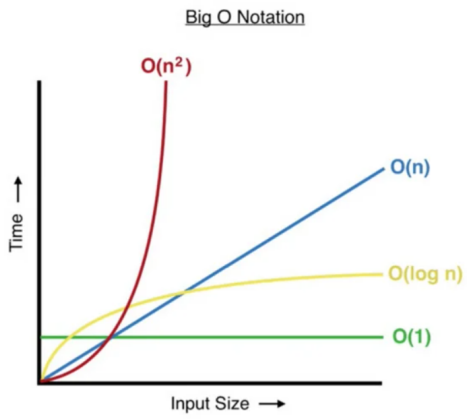


Fig. 1. Gráfico de crescimento das funções em relação ao tempo

Dessa maneira, quando comparado em complexidade de tempo, o Cocktail Sort perde em tempo de execução para alguns outros algoritmos de ordenação. Pórem, quando olhamos a complexidade de espaço, o Cocktail Sort se torna muito eficiente, pois é classificado como um in-place algorithm.

Um in-place algorithm é um algoritmo que utiliza uma quantidade de memória adicional que é independente do tamanho da entrada. Ele produz a saída na mesma área de memória que contém o array ou lista de entrada. Em outras palavras, um algoritmo in-place tem uma complexidade de espaço constante, O(1), significando que ele não requer memória extra proporcional ao tamanho da entrada. Para um algoritmo ser considerado in-place, sua complexidade de memória deve seguir uma regra de tamanho, em que tem que ser maior ou igual ao espaço constante, e menor ou igual ao espaço equivalente a O(log n).

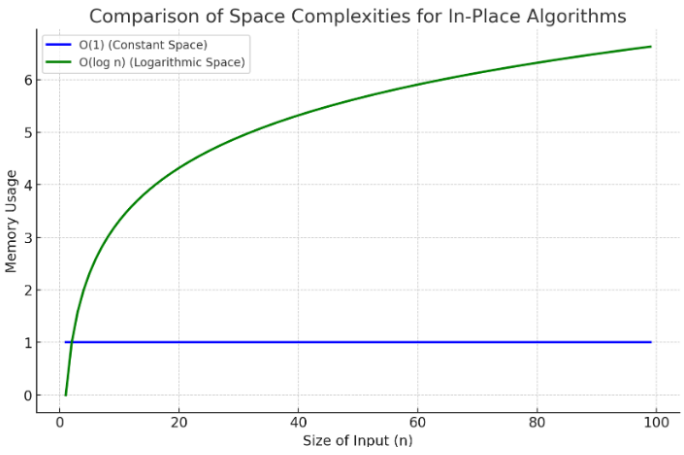


Fig. 2. Gráfico de crescimento das funções em relação a memória

Assim, para ser considerado um in-place algorithm, a linha de crescimento do algoritmo tem que estar entre a linha verde (log n) e a linha azul (espaço constante). Portanto, quando a memória é uma preocupação, o Cocktail Sort é eficiente em comparação a outros algoritmos de ordenação.

### *B. Bucket Sort*

O Bucket Sort é um algoritmo que usa uma técnica de ordenação que consiste em dividir os elementos do vetor ou lista em vários subgrupos, ou baldes (buckets), em um novo vetor de tamanho N-1. Esses baldes são formados por elementos distribuidos de maneira uniforme pelas posições do novo vetor. Para que os números sejam distribuídos nos subgrupos, é necessário pegar cada elemento e multiplicá-lo pelo números de subgrupos, para que esse elemento seja posto na posição correspondente do vetor auxiliar. Após a distribuição dos elementos, eles serão ordenados por um método de ordenação à escolha do usuário. Por último, os baldes são concatenados de maneira crescente ou decrescente.

Uma característica que diferencia o Bucket Sort de outros algoritmos de ordenação é a sua complexidade de tempo e espaço. Isso acontece devido as variações de complexidade que ele apresenta, a depender de qual algoritmo o usuário irá escolher para organizar os subgrupos, durante o processo de ordenação. Assim, o Bucket Sort possui como melhor caso, uma complexidade de O(n) + a complexidade do algoritmo escolhido pelo usuário. O melhor caso apenas acontecerá quando os elementos do vetor estarem uniformimente distribuidos pelos baldes.

Portanto, para diferentes algoritmos de ordenação escolhidos, ocorrerão diferentes complexidades de tempo.

| Sorting Algorithm | Bucket Time Complexity | Overall Time Complexity |
|---|---|---|
| Insertion Sort | $O(m^2)$ | $O(n+k+\sum_{i=1}^{k} m_i^2)$ |
| Merge Sort | $O(m \log m)$ | $O(n+k+\sum_{i=1}^{k} m_i \log m_i)$ |
| Quick Sort | $O(m \log m)$ | $O(n+k+\sum_{i=1}^{k} m_i \log m_i)$ |
| Counting Sort | $O(m+r)$ | $O(n+k+\sum_{i=1}^{k}(m_i+r_i))$ |

Fig. 3. Impacto de diferentes algoritmos de ordenação na complexidade de tempo

Analisando a tabela, fica exposto que ocorrem dois tipos de impacto na complexidade de tempo.

- Para algoritmos de ordenação por comparação, como o Merge sort e o Quick sort, a complexidade de tempo, assumindo o melhor caso, geralmente será exposta como:

$$O(nlog \times \frac{n}{k}) \tag{1}$$

- Para algoritmos de ordenação que não utilizam comparação, como o Counting sort (quando aplicável), a complexidade de tempo é mais eficiente, porém os casos em que é possível utilizar o Counting sort por exemplo, é limitado a números do tipo inteiro.

Como exemplo de complexidade de tempo, e fazendo uma interligação entre os dois algoritmos analisados, será feita uma análise de como a complexidade ficará utilizando o Cocktail Sort para ordenação dos baldes. Dessa forma, utilizando o Cocktail Sort, a comlexidade de tempo para percorrer e

ordenar cada balde será de $O(N^2)$, em que N é a quantide de números em cada balde. O processo ocorre em 3 passos:

- **Distribuição de elementos pelos baldes**. Esse processo possui custo computacional de O(n)
- **Ordenação de cada balde pelo Cocktail Sort**. Esse processo possui custo computacional de $O(N^2)$ para cada balde. Portanto, o custo completo será representado pela somatória de todos os baldes, em que m é o número de elementos em cada balde.

$$O(\sum_{n=1}^{k} m^2) \tag{2}$$

- **Combinação de todos os baldes para formar o vetor ordenado**. Esse processo possui o custo de O(n).

Assim, o custo computacional completo utilizando o Cocktail Sort para ordenar os baldes será de:

$$[O(n) + O(\sum_{n=1}^{k} m^2) \ + O(n)] = [O(n + \sum_{n=1}^{k} m^2)] \tag{3}$$

Para o caso médio, considerando que os elementos estão uniformemente distribuidos nos K baldes, cada balde terá uma quantidade de elementos gual N/K. Consequentemente, a soma dos quadrados dos números de elementos em cada balde pode ser aproximada para:

$$O(\sum_{n=1}^{k} m^2) \ = k(\frac{n}{k})^2 = \frac{n^2}{k} \tag{4}$$

Dessa maneira, a complexidade de tempo do algoritmo será de $O(n + n^2/k)$. Para números grandes de K, essa complexidade de tempo é eficiente. Porém para números pequenos de K, a divisão se torna ineficiente. Para esses casos, o custo computacional do algoritmo será de $O(n^2)$.

| Sorting Algorithm | Time Complexity (per bucket) | Overall Time Complexity |
|---|---|---|
| Cocktail Sort | $O(m^2)$ | $O(n + \sum_{i=1}^{k} m_i^2)$ |

Fig. 4. Complexidade de tempo quando usado o Cocktail Sort para ordenar os baldes

Analisando agora, a complexidade de espaço, o Bucket Sort não pode ser considerado um in-place sort algoritm, pois necessita um espaço extra proporcional ao número de elementos do vetor e ao número de baldes K. A memória adicional usada para os subgrupos excede o espaço constante que é requerido em in-place sorting algoritms.

| Algorithm | Space Complexity | In-Place? |
|---|---|---|
| Bucket Sort | $O(n + k)$ | No |

Fig. 5. Complexidade de espaço do Bucket Sort

Portanto, após as análises, fica exposto que o Bucket Sort, quando analisado a complexidade de tempo, pode ser eficiente para pequenos vetores ou listas, a depender do algoritmo de ordenação escolhido para ordenar os baldes. Já em complexidade de tempo, o algoritmo não trabalha com uma memória constante, mas sim variável e a depender da quantidade de baldes do vetor ou lista.

# III Modelos de Aplicação

As diferentes nuances de cada algoritmo fazem com que eles apresentem vantagens e desvantagens particulares.Nesta seção, será abordado os modelos de aplicação dos algoritmos Cocktail Sort e Bucket Sort, destacando-se o cenário em que cada um deles é mais eficiente e adequado.

## *A. Cocktail Sort*

O algoritmo Cocktail Sort teve sua criação a partir de um aprimoramento do Bubble Sort, fazendo com que ele se tornasse mais eficiente e adequado para determinados casos.

Expandindo a abordagem utilizida do Bubble sort, o Cocktail Sort tem seu funcionamento permitindo que a lista seja percorrida em ambas as direções em cada passagem, enquanto que o algoritmo bubble sort tem seu funcionamento avançando da esquerda para a direita, fazendo com que o maior elemento ”borbulhe” para a extremidade direita da lista

Nesse sentido, com essa possibilidade bidirecional de ordenação do algoritmo, seu uso tornou-se mais vantajoso em determinadas circunstâncias.

À priori, quando leva-se em consideração a complexidade de espaço do algoritmo com um custo O(1), esse quesito torna ele mais eficiente, utilizando de um tamanho constante indepente do tamanho da lista. Esse fator ressalta uma importante vantagem no uso do Cocktail Sort, visto que ele ordena a lista no espaço original de memória, sem precisar de espaço adicional, como mencionado anteriormente.

Além disso, o Cocktail Sort é particularmente eficiente para ordenar listas pequenas, onde sua performance é superior em comparação a outros algoritmos. Essa eficiência o torna uma escolha adequada para a ordenação de sublistas dentro de listas maiores, aproveitando sua rapidez em contextos de menor escala.

Outra vantagem significativa do Cocktail Sort é sua melhor performance em determinadas situações, especialmente quando comparado a outros algoritmos de tempo quadrático, como o Bubble Sort. Em cenários onde a lista já está ordenada ou quase ordenada, o Cocktail Sort é capaz de detectar essa condição e finalizar o processo de ordenação mais rapidamente. Isso se deve ao seu mecanismo bidirecional, que permite a detecção precoce da ordenação completa da lista.

Embora o Cocktail Sort apresente melhorias em relação ao Bubble Sort, como a ordenação bidirecional, ele possui algumas desvantagens significativas.

Primeiramente, a complexidade de tempo do Cocktail Sort é $O(n^2)$ tanto no caso médio quanto no pior caso. Isso significa que o tempo necessário para ordenar uma lista aumenta quadraticamente com o número de elementos. Para

grandes conjuntos de dados, essa característica resulta em um desempenho lento. Em comparação, algoritmos como Quick Sort e Merge Sort têm uma complexidade de tempo média de O(n log n), tornando-os mais adequados para conjuntos de dados grandes devido ao seu desempenho significativamente mais rápido.

Além disso, o Cocktail Sort requer o acompanhamento e a atualização dos índices iniciais e finais dos subarrays sendo ordenados em cada passagem. Especificamente, o algoritmo mantém e ajusta índices que delimitam a parte não ordenada do array durante as passagens bidirecionais. Esse controle extra é necessário para garantir que o algoritmo não faça comparações desnecessárias e possa adaptar sua área de operação conforme o array é ordenado. No entanto, o gerenciamento desses índices adiciona uma camada de complexidade ao algoritmo. Manter e atualizar esses índices em cada passagem pode tornar a implementação mais complexa e a execução do algoritmo mais lenta devido às operações adicionais necessárias para gerenciar e verificar esses índices. Assim, o controle adicional dos índices pode impactar negativamente a eficiência do algoritmo em termos de tempo de execução. Em contraste, algoritmos como Quick Sort e Merge Sort, que são projetados para serem mais eficientes, não precisam desse nível de gerenciamento de índices, o que contribui para seu desempenho superior.

Em resumo, o Cocktail Sort se destaca por sua eficiência espacial, com complexidade de espaço O(1), e por sua capacidade de ordenar listas pequenas de maneira eficaz. Sua habilidade de terminar prematuramente em casos onde a lista já está ordenada oferece uma vantagem adicional sobre outros algoritmos de ordenação quadrática, tornando-o uma escolha robusta e eficiente em diversas situações de ordenação. No entanto, sua complexidade de tempo de $O(n^2)$ o torna ineficiente para listas grandes. Além disso, o algoritmo requer o acompanhamento e a atualização dos índices iniciais e finais dos subarrays durante as passagens bidirecionais, o que adiciona complexidade à sua implementação e pode impactar negativamente o desempenho. Em comparação, algoritmos mais avançados, como Quick Sort e Merge Sort, oferecem melhor desempenho devido à sua complexidade de tempo média de O(n log n) e não necessitam desse nível de gerenciamento de índices. Portanto, a escolha do Cocktail Sort deve ser feita com base no tamanho e na natureza dos dados a serem ordenados.

### *B. Bucket Sort*

O Bucket Sort é um algoritmo de ordenação que organiza os dados em baldes, utilizando índices para distribuir os itens de forma uniforme. Diferente de algoritmos baseados em comparações, como o Cocktail Sort, o Bucket Sort não depende de comparações diretas entre os elementos. Em vez disso, a distribuição é baseada em índices, o que pode levar a uma ordenação eficiente quando aplicado em cenários adequados.

Uma das principais vantagens do Bucket Sort é a flexibilidade em relação aos algoritmos de ordenação que podem ser utilizados para processar os baldes individualmente. Cada balde pode ser ordenado com o algoritmo mais apropriado, dependendo da quantidade e da natureza dos dados em cada balde. Essa versatilidade permite otimizações específicas e torna o Bucket Sort aplicável a diferentes cenários de ordenação.

Além disso, o Bucket Sort tem uma vantagem quando se utiliza da sua capacidade de aproveitar o paralelismo para melhorar o desempenho.O paralelismo consiste em uma técnica na computação, na qual várias instruções são executadas simultaneamente, diminuindo dessa maneira o tempo de resolução de um problema.Um problema maior é dividido em subproblemas e esses subproblemas são executados em paralelo nos múltiplos núcleos ou threas, que executam eles de maneira independente.

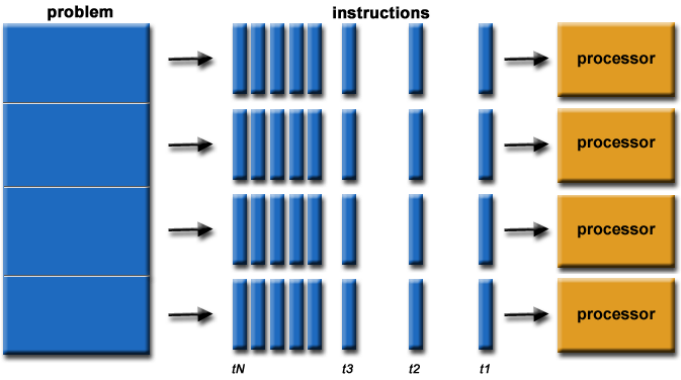


Fig. 6. Esquema da resolução de um problema com Paralelismo

De maneira análoga, na versão paralela do Bucket Sort, é possível utilizar múltiplos threads para ordenar os baldes simultaneamente. Em um ambiente de computação paralela, diferentes threads ou processos podem ser designados para trabalhar em diferentes baldes ao mesmo tempo, o que pode acelerar significativamente a fase de ordenação, especialmente em sistemas com múltiplos núcleos de CPU ou em ambientes de computação distribuída. Esse paralelismo permite que cada balde seja tratado de forma independente, otimizando o desempenho e reduzindo o tempo total de ordenação.

Apesar das vantagens, o Bucket Sort também tem suas desvantagens. A eficiência do algoritmo é sensível à distribuição dos valores de entrada. Se os valores estiverem muito agrupados ou distribuídos de forma não uniforme, o algoritmo pode acabar realizando uma quantidade excessiva de trabalho sem obter um ganho significativo de desempenho. Além disso, o desempenho do Bucket Sort pode depender do número de baldes escolhidos, o que pode exigir ajustes para otimizar a eficiência.

Em suma, o Bucket Sort é um algoritmo poderoso e flexível que pode ser altamente eficiente quando aplicado a conjuntos de dados com distribuição uniforme. Suas capacidades de paralelismo e flexibilidade no uso de diferentes algoritmos para ordenar baldes são grandes vantagens. No entanto, a escolha do número de baldes e a consideração da distribuição dos dados são cruciais para maximizar o desempenho do algoritmo.

# IV Resultados

A avaliação dos algoritmos Cocktail Sort e Bucket Sort foi conduzida de forma a permitir uma análise detalhada de seu desempenho em diferentes condições. Os testes foram realizados em vetores desordenados, com variados tamanhos de entrada (1000,10.000,100.000,500.000). Adicionalmente, considerou-se o desempenho em linguagens compiladas - C++, C e C# - e interpretadas - Java, Javascript, e Python. Essa abordagem multifacetada permitiu analisar como diferentes fatores influenciam o tempo de execução de cada algoritmo e avaliar seu desempenho em diversos contextos.

Para a testagem dos algoritmos foi utilizado uma máquina com Windows 11, processador 11th Gen Intel Core i7-11390H 3.40GHz, 1 CPU, 8 logical and 4 physical cores e 16Gb de memória ram. Além disso, para que se obtivesse resultados mais precisos os algoritmos foram rodados em média de cinquenta a cem vezes e assim calculou-se o tempo médio de execução para os diferentes tamanhos de entrada.

Mediante a isso, os pesudocódigos dos algoritmos Cocktail Sort e Bucket Sort ilustram de maneira clara o funcionamento de cada um deles, bem como as etapas e lógica dos mesmos.

Abaixo, é representado o pseudocódigo do Cocktail Sort, que mostra sua implementação e sua lógica de ordenação, bem como a evidenciação do seu custo quadrático.

**Algorithm 1** Cocktail Sort

```
1:  procedure CocktailSort(A)
2:      n ← comprimento(A)
3:      trocou ← Verdadeiro
4:      inicio ← 0
5:      fim ← n − 1
6:      while trocou do
7:          trocou ← Falso
8:          for i ← inicio até fim − 1 do
9:              if A[i] > A[i + 1] then
10:                 trocar(A[i], A[i + 1])
11:                 trocou ← Verdadeiro
12:             end if
13:         end for
14:         if ¬trocou then
15:             quebrar
16:         end if
17:         trocou ← Falso
18:         fim ← fim − 1
19:         for i ← fim − 1 até inicio decrescente do
20:             if A[i] > A[i + 1] then
21:                 trocar(A[i], A[i + 1])
22:                 trocou ← Verdadeiro
23:             end if
24:         end for
25:         inicio ← inicio + 1
26:     end while
27: end procedure
```

Agora, será apresentado o pseudocódigo do Bucket Sort, sua implementação e sua lógica de ordenação por baldes. O código mostra a implementação apenas do algoritmo principal, ou seja, não possui a pseudo implementação do insertion sort, usado para ordenar os baldes.

**Algorithm 2** Bucket Sort

```
1:  Input: Array A of n elements
2:  Output: Sorted array A
3:  procedure BucketSort(A)
4:      n ← length(A)
5:      B ← array of n empty buckets
6:      maxValue ← maximum value in A
7:      minValue ← minimum value in A
8:      for each element a in A do
9:          index ← ⌊(a − minValue)/(maxValue − minValue + 1) × n⌋
10:         B[index].append(a)
11:     end for
12:     for each bucket b in B do
13:         Sort(b)
14:     end for
15:     index ← 0
16:     for each bucket b in B do
17:         for each element a in b do
18:             A[index] ← a
19:             index ← index + 1
20:         end for
21:     end for
22: end procedure
```

As tabelas abaixo mostram as linguagens em que os códigos foram implementados, os diferentes tamanhos de entrada e as médias dos tempos de execução:

TABLE I
TEMPOS DE EXECUÇÃO EM MS DO ALGORITMO COCKTAIL SORT EM DIFERENTES LINGUAGENS

| Tamanho da Entrada | Python | Java | JavaScript | C | C++ | C# |
|---|---|---|---|---|---|---|
| 1000 | 36.09 | 7.00 | 56.05 | 2.21 | 1.93 | 0.46 |
| 10000 | 3668.09 | 142.00 | 101.36 | 269.16 | 212.21 | 43.54 |
| 100000 | 422483.15 | 10687.00 | 13486.89 | 29848.88 | 250900 | 8393.48 |
| 500000 | - | 333186.00 | 368887.25 | 720146.19 | 641280 | 212662.70 |

TABLE II
TEMPOS DE EXECUÇÃO EM MS DO ALGORITMO BUCKET SORT EM DIFERENTES LINGUAGENS

| Tamanho da Entrada | Python | Java | JavaScript | C | C++ | C# |
|---|---|---|---|---|---|---|
| 1000 | 0.36 | 0.39 | 55.90 | 0.12 | 0.0015 | 0.02 |
| 10000 | 6.01 | 1.12 | 59.49 | 6.29 | 0.0058 | 0.48 |
| 100000 | 76.51 | 6.67 | 85.20 | 1372.89 | 0.084 | 20.36 |
| 500000 | 463.35 | 67.12 | 192.62 | 102483.67 | 0.39 | 99.72 |

### *A. Resultados dos algoritmos em Python*

A primeira leva de testes feita com os algoritmos foi realizada na linguagem interpretada python. Com a execução, obteve-se os resultados mostrados nos gráficos abaixo. Os

gráficos mostram como o tempo médio de execução muda a medida que os tamanhos de entrada aumentam.

O algoritmo Cocktail Sort não suportou vetores com entradas de tamanho 500.000. Ele foi executado apenas até tamanhos de vetores com 100.000 números. Ao receber vetores de entrada com tamanho de 500.000, o programa permaneceu em execução sem gerar nenhum resultado de tempo médio de execução. A linguagem Python foi a única que não conseguiu executar o algoritmo Cocktail Sort com todos os tamanhos de entrada.

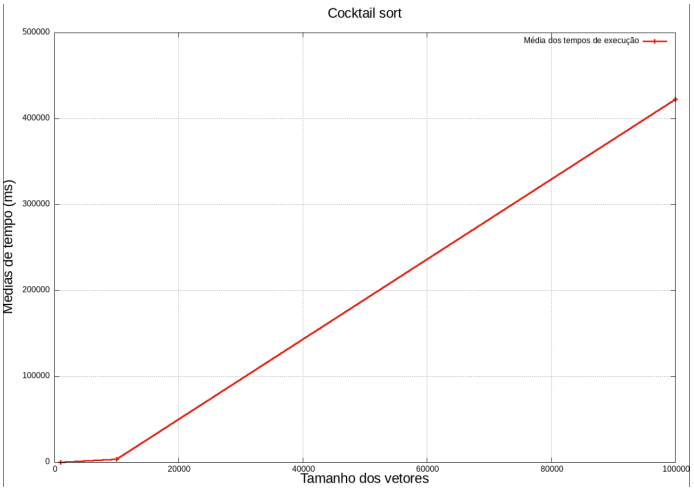


Fig. 7. Gráfico do tempo de execução do Cocktail Sort em Python

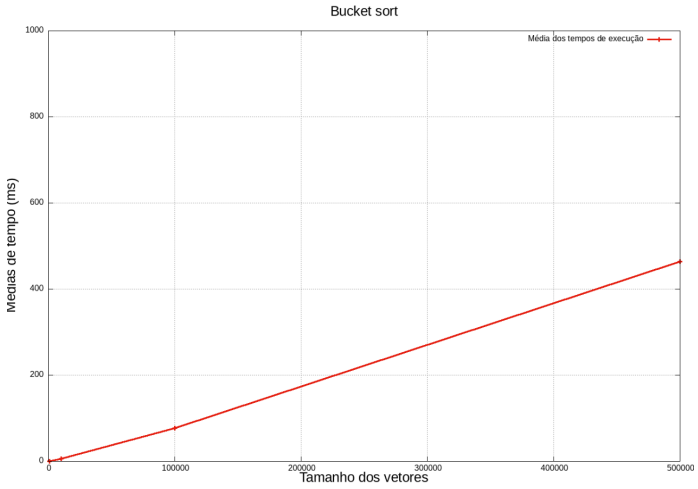


Fig. 8. Gráfico do tempo de execução do Bucket Sort em Python

Em contrapartida, o Bucket Sort executou todos os tamanhos de entrada, demonstrando uma eficiência significativa em relação ao Cocktail Sort, obtendo um tempo médio muito mais ágil.

A comparação com as outras linguagens mostrou que em python o tempo médio de execução foi maior em comparação com as linguagens compiladas.

### *B. Resultados dos algoritmos em JavaScript*

Os gráficos abaixo mostram os tempos médios de execução dos algoritmos na linguagem JavaScript, a qual, assim como python, também é interpretada.

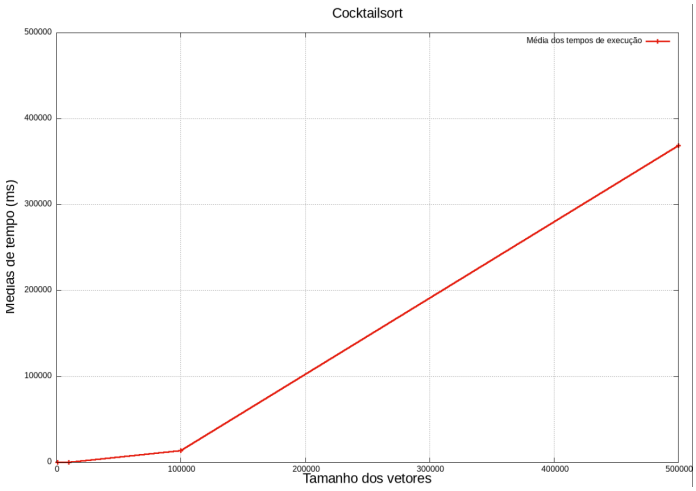


Fig. 9. Gráfico do tempo de execução do Cocktail Sort em Javascript

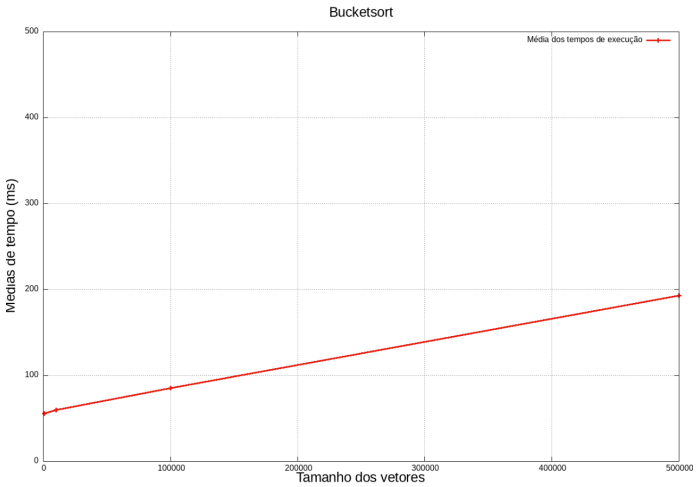


Fig. 10. Gráfico do tempo de execução do Bucket Sort em Javascript

Ao se analisar os gráficos pode-se perceber que para entradas de tamanho 1000, ambos os algoritmos apresentaram tempos médio de execução muito próximos, entretanto a medida que o tamanho das entradas aumentam, o tempo gasto pelo Cocktail Sort é muito maior, demonstrando uma certa ineficiência desse algoritmo.

Ao se comparar com outras liguagens, o Javascript fica atrás de algoritmos que são compilados gastando um tempo maior para ordenar os vetores. Em comparação com os algoritmos interpretados o JavaScript teve um desempenho melhor que o python quando foi testado o Cocktail Sort, conseguindo ordenar os vetores até entradas de tamanho de 500000, mas quando testado com o Bucket Sort, o algoritmo em JavaScript teve um desempenho inferior.

### C. Resultados dos algoritmos em Java

A implementação dos algoritmos para testagem também foi realizada em Java. Essa linguagem é de certa forma uma linguagem híbrida, pois o código é compilado para bytecode pela JVM, que é a máquina virtual do java, e esse bytecode gerado é interpretado para código nativo pelo Jit compiler durante a execução. Os gráficos abaixo mostram o tempo médio de execução dos algoritmos para cada tamanho de entrada.

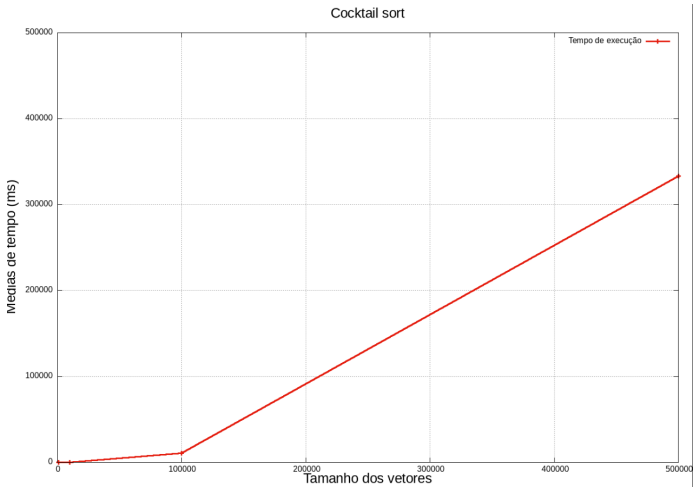


Fig. 11. Gráfico do tempo de execução do Cocktail Sort em Java

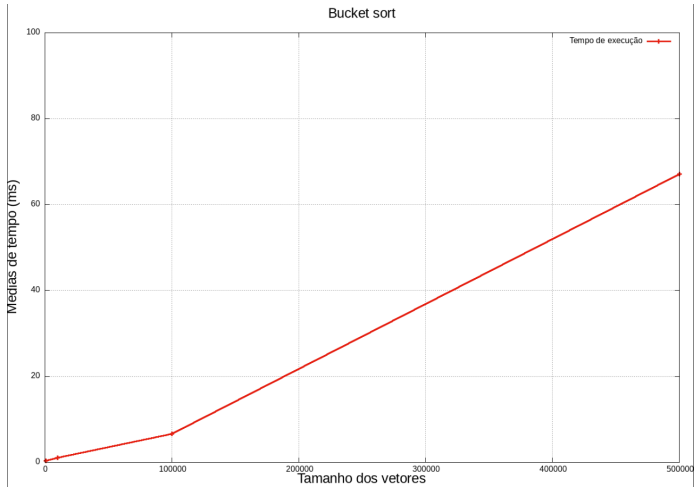


Fig. 12. Gráfico do tempo de execução do Bucket Sort em Java

Por ser uma linguagem interpretada, Java em teoria deveria ter um tempo de execução maior que as linguagens compiladas. Entretanto, nem todos os resultados mostraram isso. Através da utilização da JVM, essa máquina virtual acaba gerenciando a memória e a execução de threads, oferecendo recursos como coleta de lixo e o JIT(Just in time compilation), os quais podem ter contribuído significativamente para otimizações no tempo de execução.

### D. Resultados dos algoritmos em C

A linguagem C foi a primeira das linguagens compiladas testadas. Os gráficos abaixo mostram como o tempo de execução muda a medida que o tamanho das entradas aumentam:

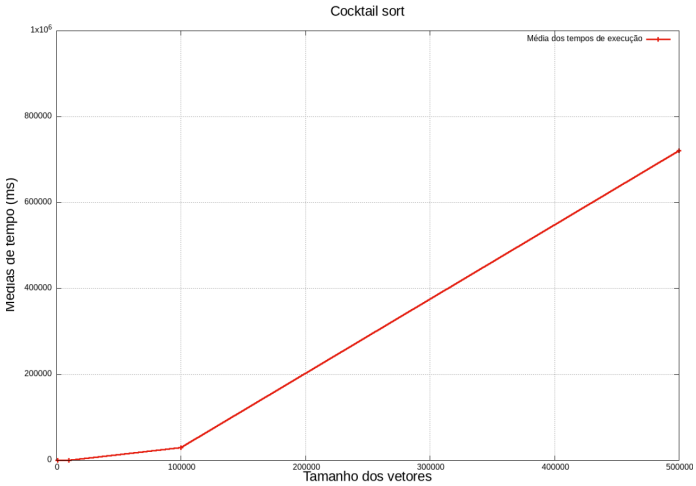


Fig. 13. Gráfico do tempo de execução do Cocktail Sort em C

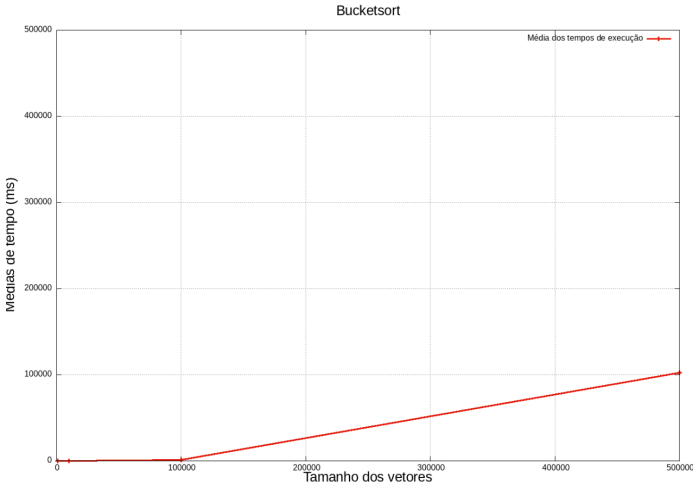


Fig. 14. Gráfico do tempo de execução do Bucket Sort em C

Fazendo um comparativo com outras linguagens compiladas, C teve um desempenho ruim e se mostrou ineficiente para vetores com grandes tamanhos de dados a serem ordenados.

### E. Resultados dos algoritmos em C++

Para os gráficos em C++, são representados dois tempos de execuções, um representando o tempo real ( real execution time) e o outro representado o tempo de execução de CPU. Dessa maneira, observa-se que o tempo de execução da CPU é menor e mais preciso do que o tempo real.

Essa diferenciação ocorre porque o tempo real de execução não inclui apenas o tempo que a CPU gasta executando as

instruções do programa, mas também o tempo gasto esperando outros recursos do sistema. Por outro lado, o tempo de execução da CPU mede apenas o tempo gasto para executar as instruções do programa, por isso ele é mais preciso e menor em comparação ao retal execution time.

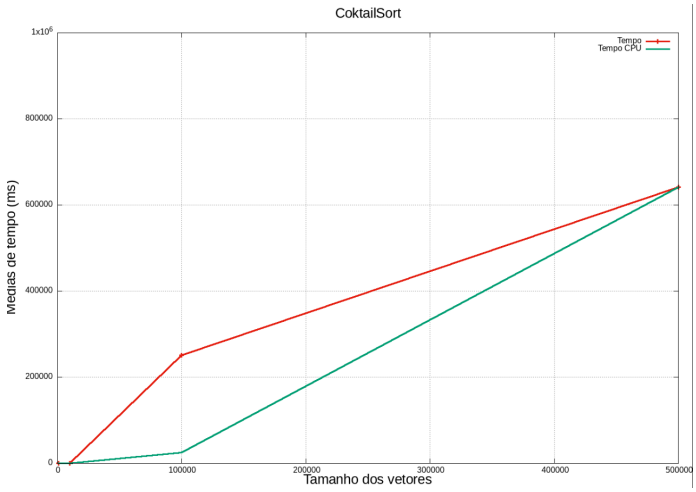


Fig. 15. Gráfico do tempo de execução do Cocktail Sort em C++

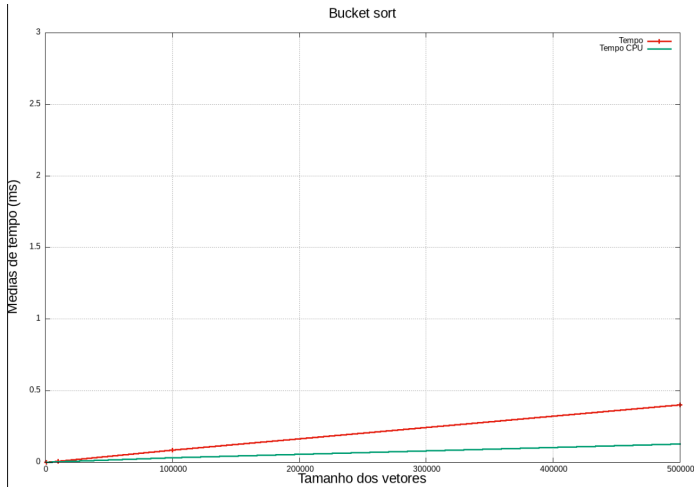


Fig. 16. Gráfico do tempo de execução do Bucket Sort em C++

Com base na implementação usando paralelismo do Bucket Sort, descrita na seção III, pode-se evidenciar, por meio dos resultados, o melhor desempenho do Bucket Sort com o uso do recurso.

### *F. Resultados dos algoritmos em C#*

Por fim, os algoritmos foram testados na linguagem compilada C#. Os gráficos abaixo ilustram os tempos médios de execução para diferentes tamanhos de entrada.

A análise dos algoritmos em C# demonstrou que o tempo de execução médio entre o Cocktail Sort e o Bucket Sort apresentaram notáveis diferenças. Comparando os dados, observa-se que o algoritmo Bucket Sort foi consideravelmente melhor em todos seus tempos de execução. Entretanto, mesmo com

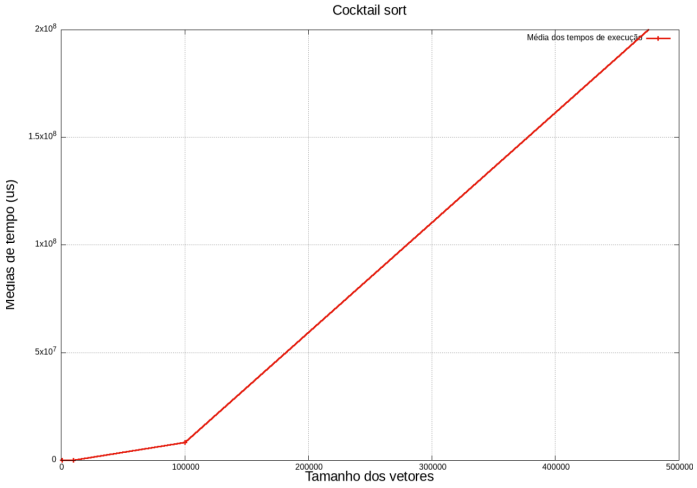


Fig. 17. Gráfico do tempo de execução do Cocktail Sort em C#

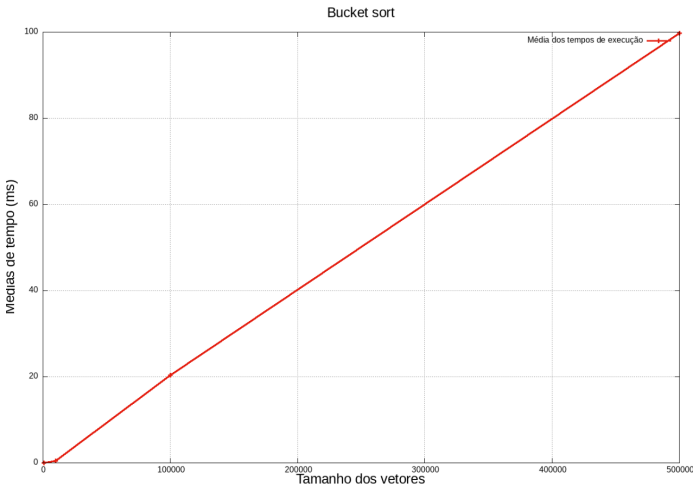


Fig. 18. Gráfico do tempo de execução do Bucket Sort em C#

um desempenho pior que o Bucket Sort, o Cocktail Sort teve resultados melhores quando comparado com as outras linguagens testadas com esse algoritmo.

## V Insights

Ao decorrer do trabalho, alguns questionamentos foram levantados sobre as funcionalidadades do código e sobre a teoria da computação em geral. Sendo assim, nessa seção serão tratados esses tópicos, apresentando as dúvidas, os recursos utilizados para saná-las, e suas respostas.

### *A. Orientação à objetos x Programação Procedural*

Inicialmente, os códigos dos algoritmos em C++ foram implementados usando programação procedural. Porém, após a testagem dos códigos e análise das medições, ocorreu a reimplementação dos algoritmos utilizando Orientação à objetos, por meio de classes que possum os nomes dos algoritmos em análise - Cocktail Sort e Bucket Sort.

Diante disso, foi observado um leve aumento no tempo de execução dos códigos, representado por alguns milisegundos de diferença. Ao analisar esse aumento, foi levantada a seguinte questão:

*Usar Orientação a Objetos causa mais overhead no código?*

A orientação a objetos pode causar alguns overheads específicos, dentre eles:

- **Alocação de memória** ao instanciar objetos da classe;
- **Encapsulamento e abstração de métodos** ao usar getters e setters ao invés de acessar diretamenteos membros da classe;
- **Polimorfismo** por meio de tabelas virtuais e resolução dinâmica ( o que não é o caso dos algoritmos utilizados).

Analisando tais fatores que podem causar possíveis overheads é comum afirmar-se que a Orientação à objetos aumenta o tempo de execução do código. Entretanto, especialmente para c++, há uma lei que postula sobre o overhead: The Zero-overhaed principle. Tal postulado expressa que o que não é usado em código, não é pago em tempo, espaço ou algo mais profundo. Assim, todo overhead causado pela orientação à objetos não será cobrado no tempo de execução do código.

Desse modo, segundo o Cientista da Computação Bjarne Stroustrup, considerado o pai de c++:

> ”If you have an abstraction, it should not cost anything compared to writing the equivalent code at a lower level. So, if I have a matrix multiplier, it should be written in such a way that you could not drop to C level of abstraction and use arrays and pointers and run faster.”
>
> – Bjarne Stroustrup

Comprovando o que foi dito pelo cientista, o usuário do GitHub `@baderouaich` realizou experimentos em que ele implementava o mesmo código em C - utilizando programação procedural - e em C++ - utilizando Oriantação á Objetos - e verificou que as movimentações em Assembly eram as mesmas para as duas linguagens, ou seja, não ocorreu o aumento do overhead pelo uso de Orientação à Objetos.

```
main:
        movl    $18620, %eax
        retq
```

Fig. 19. Número de movimentaçãos Assembly para C e C++

Portanto, reforçando ainda mais o Zero-overhead principle, o Cientista da Computação Bjarne Stroustrup discorre sobre a tradução dos códigos para linguagens de baixo nível e como isso não afetará as movimentações em Assembly:

> ”It doesn’t mention it’s absolutely optimal, but it means if you’re hand coded either the usual facilities in the language C++ and C you should not be able to better it.”
>
> – Bjarne Stroustrup

Por fim, fica eviente que o uso de características avançadas da linguagem, como templates, classes, funções inline, não deve resultar em desempenho inferior ao do código otimizado escrito manualmente.

### *B. Linguagens compiladas x Executadas*

Em teoria linguagens compiladas tem um tempo de execução mais rápido que interpretadas. Esse fator se deve principalmente ao fato de que nas compiladas o código fonte é traduzido diretamente para código de máquina antes da execução, eliminando a necessidade de tradução em tempo real, enquanto que nas interpretadas o código é traduzido linha por linha durante a execução do programa.

Entretanto, como analisado anteriormente, durante os testes pode-se perceber que Java, uma linguagem que em partes é interpretada, apresentou tempos de execução melhores que algumas compiladas, como o C. Esses resultados se devem, principalmente, a avanços como a compilação Just in time em linguagens interpretadas, que otimiza o bytecode em tempo de execução, gerando um código altamente otimizado para a máquina.

Além disso, outros fatores podem ter contribuído para resultados melhores de Java em comparação com algumas linguagens compiladas. A JVM, por exemplo, pode gerar otimizações para o uso do hardware específico, como também pode proporcionar um melhor gerenciamento de memória e uma coleta de lixo, sendo esses fatores contribuintes para melhora no tempo de execução.

## VI Conclusão

Ao longo deste artigo, foram explorados em detalhes os algoritmos de ordenação Cocktail Sort e Bucket Sort, destacando suas características, vantagens, desvantagens e cenários ideais de aplicação. A análise detalhada de cada algoritmo revelou insights valiosos sobre como a escolha do método de ordenação pode impactar significativamente o desempenho, especialmente em termos de tempo de execução e consumo de memória.

O Cocktail Sort, uma variação do Bubble Sort, demonstrou ser uma escolha eficiente para conjuntos de dados menores e casos onde a memória é uma preocupação, devido à sua complexidade espacial constante (O(1)). No entanto, sua complexidade temporal ($O(n^2)$) limita sua eficiência para conjuntos de dados maiores, tornando-o menos adequado para aplicações em grande escala.

O gráfico a seguir representa a comparação entre todas as linguagens analisadas:

Analisando o gráfico, têm-se que o melhor tempo de execução foi representado pela linguagem C#, e o pior tempo de execução pela linguagem Python, a qual não suportou rodar o vetor desordenado de 500.000 posições.

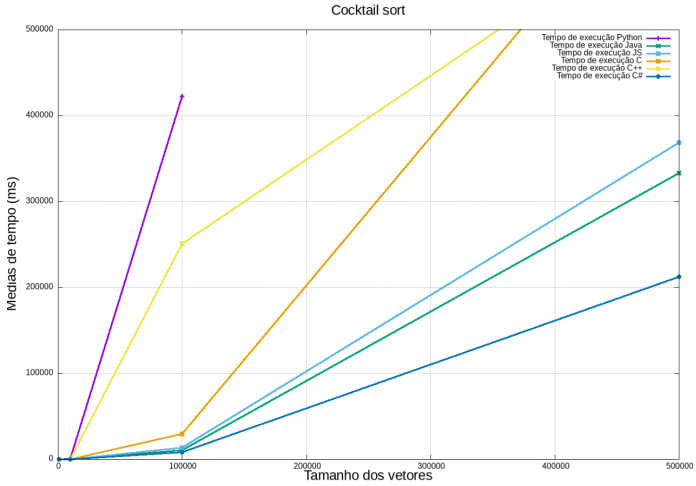


Fig. 20. Grafíco Cocktail Sort para todas as linguagens

Por outro lado, o Bucket Sort se mostrou altamente flexível e eficiente em situações onde os dados são distribuídos uniformemente. Sua capacidade de se beneficiar do paralelismo e de utilizar diferentes algoritmos de ordenação para os baldes oferece uma vantagem significativa em termos de tempo de execução.

No entanto, a necessidade de memória adicional para armazenar os baldes impede que ele seja considerado um algoritmo in-place, e sua eficiência pode variar dependendo da distribuição dos dados e do número de baldes utilizados.

O gráfico a seguir representa a comparação entre todas as linguagens analisadas:

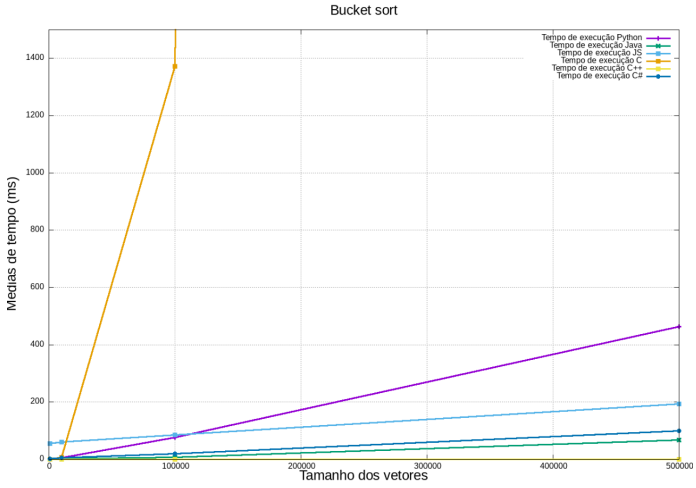


Fig. 21. Gráfico Bucket Sort para todas as linguagens

Analisando o gráfico, têm-se que o melhor tempo de execução foi representado pela linguagem C++ (em que foi usado o Bucket Sort paralelo), e o pior tempo de execução pela linguagem C, a qual se destoou no tempo de execução de vetores grandes - 100.000 e 500.000 posições.

# References


[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.

[2] MANASEER, Saher; AL HWAITAT, Ahmad K. Measuring Parallel Performance of Sorting Algorithms: Bubble Sort and Bucket Sort on IMAN 1. Mathematical and Computational Applications, [S.l.], v. 12, n. 10, p. 23, 2018. Disponível em: Peformance of Sorting Algorithms: Bubble Sort and Bucket Sort on IMAN 1. Acesso em: 15 jul. 2024.

[3] STROUSTRUP, Bjarne. C++ exceptions and alternatives P1947. Doc. No. P1947, 2019. Disponível em: C++ exceptions and alternatives . Acesso em: 15 jul. 2024.

[4] baderouaich. the-zero-overhead-principle. Doc. No. 2023. Disponível em: the-zero-overhead-principle. Acesso em: 15 jul. 2024.