# Ordenações Bidirecionais e Distribuídas

Danniel Vieira Holanda
*Engenharia de Computação*
*CEFET - MG*
Divinópolis, Brasil
dannielvh7@gmail.com

Igor Moreira Lopes
*Engenharia de Computação*
*CEFET - MG*
Divinópolis, Brasil
igor.moreira.lopes22@gmail.com

Kairo Henrique Ferreira Martins
*Engenharia de Computação*
*CEFET - MG*
Divinópolis, Brasil
kairohenrique293@gmail.com

Tauane Luisa Silva
*Engenharia de Computação*
*CEFET - MG*
Divinópolis, Brasil
tauanealeatorias@gmail.com

***Abstract*—This paper presents an in-depth study of three less conventional sorting algorithms: *Cocktail Sort*, *Bucket Sort*, and *Pigeonhole Sort*. Each of them presents distinct approaches to the sorting problem, either through bidirectional scans or by distributing the elements into auxiliary structures. *Cocktail Sort* is a variation of *Bubble Sort* with two-way traversal. *Bucket Sort* groups the data into buckets before sorting them locally. *Pigeonhole Sort* is efficient when the data set is limited and well distributed. Their complexities, functioning, advantages, and limitations are analyzed, with implementations in different languages and data structures. Practical tests used the `ratings.csv` dataset — found in [1] — allowing to evaluate the real behavior of each algorithm in different contexts.**

***Index Terms*—Sorting Algorithms, Cocktail Sort, Bucket Sort, Pigeonhole Sort, Complexity, Data Structures, Performance Comparison, Execution Timer, Memory Usage.**

## I. Introdução

A ordenação de dados é uma tarefa essencial na computação, com aplicações que vão desde simples organizações/ordenações, até pré-processamentos para algoritmos de busca, compressão, aprendizado de máquina e bancos de dados. Dada sua importância, uma grande variedade de algoritmos de ordenação foi desenvolvida ao longo das décadas, cada um com características específicas de desempenho e aplicabilidade.

Este trabalho tem como foco três algoritmos classificados como bidirecionais ou distribuídos: *Cocktail Sort*, *Bucket Sort* e *Pigeonhole Sort*. O *Cocktail Sort* melhora o desempenho do tradicional *Bubble Sort* ao realizar passagens nos dois sentidos, movendo simultaneamente os maiores e menores elementos para suas posições corretas. O *Bucket Sort*, por sua vez, distribui os elementos em vários "baldes", ordenando cada balde individualmente, o que pode reduzir significativamente o tempo total de execução, especialmente com dados uniformemente distribuídos. Por fim, o *Pigeonhole Sort* baseia-se na ideia de que os elementos podem ser mapeados diretamente para "casas" indexadas de acordo com seus valores, tornando possível ordenar em tempo linear, desde que certos critérios sejam cumpridos.

O objetivo deste estudo é apresentar uma análise detalhada desses algoritmos, tanto do ponto de vista teórico quanto prático, com implementações comparativas em diversas linguagens e estruturas de dados, utilizando o conjunto real `ratings.csv` [1] como base experimental.

## II. Metodologia

### A. Cocktail Sort

O *Cocktail Sort* funciona de maneira similar à do *Bubble Sort*, sua grande diferença seria a ordenação de maneira bilateral. Durante a primeira etapa do método, as comparações são feitas entre os objetos adjacentes de maneira que se o item da esquerda for maior do que o da direita, então os objetos serão trocados. Assim, após o final da primeira iteração da ordenação, o objeto de maior valor estará localizado já ao final do vetor. Partindo para a segunda etapa, o laço funcionaria em um sentido contrário, os objetos adjacentes continuam sendo trocados, mas agora o objeto de menor valor é o alvo, sendo alocado ao final da iteração no início do vetor. A lógica da implementação continua partindo do início até o final e logo depois seguindo o movimento contrário, até que o vetor esteja completamente organizado.

Considerando um vetor desordenado, na primeira passagem o primeiro objetivo é levar o objeto de maior valor até o final do array, iniciando a partir do item inicial, ele é constantemente comparado com os itens adjacentes, enquanto ele for maior do que o item de comparação ele será levado mais adiante no vetor. Seguindo esse raciocínio, quando ele alcançar um item que for maior do que ele, o item que será movimentado será aquele que barrou a movimentação do item anterior, esse loop ocorre até que o objeto de maior valor seja colocado na última posição do array. Próxima etapa segue um caminho semelhante, mas dessa vez caminhando na direção oposta, ou seja, indo do final para o início do vetor, levando agora o item de menor valor para o início. Assim, após ir e voltar, os itens com menor e maior valor já estão devidamente posicionados. Esse processo se repete quantas vezes forem necessárias até que o vetor esteja inteiramente

ordenado, não sendo mais necessário fazer nenhuma alteração entre os objetos.

O cálculo de complexidade deste método seria $O(n^2)$ considerando os seus piores casos e seus médios casos, normalmente o *Cocktail Sort* funcionaria pouco melhor do que o Bubble Sort. Em um caso de comparação básico entre os dois métodos, o *Bubble Sort* teria que caminhar quatro vezes no vetor para que a ordenação fosse concluída, enquanto o *Cocktail* teria que caminhar duas vezes completas, considerando ida e volta no vetor. Não existe diferença entre o resultado da análise assintótica, mesmo quando o método do *Cocktail* é aplicado em lista, fila e pilha, continua apresentando o mesmo desenvolvimento para todos os casos. Seu custo passa a ser a $O(n)$ quando considerado o melhor caso, um caso consideravelmente mais raro, onde o vetor já esteja totalmente ordenado

### *B. Bucket Sort*

O método do *Bucket Sort* segue uma ideia de dividir os elementos em grupos, ou baldes, depois que já tiverem sido devidamente divididos, são organizados usando outro método de ordenação e finalmente posicionados no vetor novamente, dessa vez agora ordenados.

A partir do vetor inicial, no qual estão os objetos a serem ordenados, são criados um número n de baldes, onde esse valor n é o número de itens no vetor inicial. Cada elemento do primeiro array será multiplicado pelo número n de baldes existentes, depois será convertido para um número inteiro e assim ser obtido o balde no qual ele será alocado. Esse mesmo processo é repetido para todos os elementos do vetor inicial. Após esse processo de separação, outro algoritmo de ordenação é utilizado para que todos os elementos presente num determinado balde fiquem ordenados, assim, com os baldes já ordenados de maneira interna eles serão "esvaziados", colocando novamente cada item, já ordenados, no vetor original. Quando um item dos baldes é colocado em sua devida posição no array original, ele é apagado do array auxiliar, o processo é repetido até que todos tenham sido posicionados. Ao fim, tudo estará ordenado.

No caso de complexidade do tempo deste método, quando considerado o seu pior caso, onde todos os elementos fossem colocados em um único balde, o algoritmo de ordenação auxiliar que fosse utilizado teria que ordenar tudo de uma vez de qualquer maneira. Porém, mesmo no pior caso existem muitas variações, já que os resultados dessa análise dependem quase inteiramente do algoritmo auxiliar utilizado, se usado *Heap Sort* em vetores pequenos ele poderia funcionar até como $O(n*log(n))$. Já no seu melhor caso, quando os baldes recebem um número igual de elementos, e então nesse caso toda vez que o algoritmo auxiliar de ordenação for chamado, irá funcionar com um tempo constante para ordenar todos, dessa maneira, teria uma complexidade de tempo equivalente à $O(n + k)$, tendo que k seria esse tempo constante que foi utilizado para a ordenar cada balde.

### *C. Pigeonhole Sort*

O *Pigeonhole Sort* é um algoritmo de ordenação que se baseia no princípio conhecido como "caixa de pombos", no qual os elementos são distribuídos em compartimentos de acordo com seus respectivos valores. A ideia por trás desse método é criar uma estrutura de compartimentos consecutivos — também chamados de "potes" ou "pombais" — onde cada valor possível dentro do intervalo tem sua própria posição definida. Depois da etapa inicial de distribuição, os elementos são retirados dessas caixas seguindo a ordem crescente dos valores, o que garante que fiquem organizados corretamente no vetor final. Diferente de algoritmos como o *Bubble Sort* ou o *Quick Sort*, o *Pigeonhole* não compara diretamente os elementos entre si. Em vez disso, ele trabalha com a ideia de posicionamento direto, o que o torna um exemplo de algoritmo de ordenação que não depende de comparações.

A análise de complexidade desse algoritmo é classificada como $O(n + k)$, sendo n o número de elementos presentes no vetor e k o intervalo de valores possíveis — ou seja, a diferença entre o maior e o menor elemento. Isso faz com que o *Pigeonhole* seja mais eficiente em situações em que o total de elementos e a variedade de valores que eles podem assumir estão em proporções semelhantes. Por outro lado, quando o intervalo é muito maior que o número de elementos, o desempenho do algoritmo tende a piorar, já que tanto o tempo quanto o uso de memória aumentam consideravelmente, podendo chegar perto de $O(k)$. Em compensação, quando o intervalo é pequeno e os dados são inteiros, o algoritmo funciona mais rapidamente. Uma diferença importante entre ele e o *Counting Sort* é que, enquanto este último armazena apenas a contagem de ocorrências, o *Pigeonhole* guarda os próprios elementos dentro dos compartimentos. Por isso, há duas movimentações envolvidas: uma quando os elementos são colocados nos potes, e outra quando são recolocados no vetor final. Esse comportamento garante que o algoritmo seja estável, ou seja, a ordem relativa entre elementos iguais é preservada.

Se considerarmos um vetor desorganizado, o *Pigeonhole Sort* começa identificando os valores mínimo e máximo presentes nele. Com essas informações, calcula-se quantos compartimentos serão necessários. Depois disso, cada elemento do vetor é direcionado para sua posição correspondente no array de compartimentos, com base no seu valor. Por exemplo, se o menor valor for 5 e encontrarmos o número 7 no vetor, ele será colocado na posição 2, já que 7 - 5 = 2. Assim que todos os elementos estiverem alocados corretamente, o próximo passo é percorrer todos os compartimentos, do menor para o maior índice, retirando seus conteúdos e inserindo-os de volta no vetor original. Como os elementos foram agrupados conforme seu valor e são retirados em ordem crescente, o vetor final sairá automaticamente ordenado. Esse processo é direto e ocorre uma única vez, sem necessidade de múltiplas passagens ou trocas sucessivas. Por isso, quando aplicado nas condições ideais, o *Pigeonhole Sort* mostra-se um algoritmo eficiente, de fácil entendimento e com boa estabilidade.

## III. Pseudocódigos

### A. Cocktail Sort

**Algorithm 1:** Cocktail Sort

```
Input: Array A de n elementos
Output: Array A ordenado
Function CocktailSort(A):
    n ← tamanho de A ;
    if n ≤ 1 then
        return;
    end
    incio ← 0 ;
    fim ← n − 1 ;
    trocado ← verdadeiro ;
    while trocado do
        trocado ← falso ;
        for i ← incio to fim − 1 do
            if A[i] > A[i + 1] then
                Trocar A[i] com A[i + 1] ;
                trocado ← verdadeiro ;
            end
        end
        if não trocado then
            break ;
        end
        fim ← fim − 1 ;
        for i ← fim incio + 1 do
            if A[i − 1] > A[i] then
                Trocar A[i − 1] com A[i] ;
                trocado ← verdadeiro ;
            end
        end
        incio ← incio + 1 ;
    end
```

**Funcionamendo do Cocktail Sort:** Funciona como uma versão bidirecional do *Bubble Sort*. O processo começa definindo duas fronteiras, início e fim, que marcam a porção não ordenada do array. O algoritmo então entra em um laço que continua enquanto trocas forem realizadas. Dentro do laço, ocorrem duas passadas. A primeira é uma passada para frente, da esquerda para a direita, que funciona exatamente como o *Bubble Sort*: compara elementos adjacentes e move o maior valor para o final da seção não ordenada. Ao final desta passada, a fronteira fim é decrementada, pois o maior elemento já está em sua posição correta. Em seguida, ocorre a segunda passada, no sentido inverso (da direita para a esquerda), que compara elementos adjacentes e move o menor valor para o começo da seção não ordenada. Ao término, a fronteira início é incrementada. Este ciclo de "sacudir" o array para frente e para trás se repete, encolhendo as fronteiras, até que uma passagem completa (ida e volta) ocorra sem nenhuma troca, indicando que o array está totalmente ordenado.

### B. Bucket Sort

**Algorithm 2:** Bucket Sort

```
Input: Array A de n elementos
Output: Array A ordenado
Function BucketSort(A):
    n ← tamanho de A ;
    if n ≤ 1 then
        return;
    end
    maxValue ← maior valor em A ;
    minValue ← menor valor em A ;
    if minValue == maxValue then
        return;
    end
    B ← array de n baldes vazios ;
    foreach elemento a em A do
        index ← ⌊ (a − minValue) / (maxValue − minValue) × (n − 1) ⌋ ;
        Inserir a no balde B[index] ;
    end
    foreach balde b em B do
        Ordenar b (ex: com Insertion Sort) ;
    end
    index ← 0 ;
    foreach balde b em B do
        foreach elemento a em b do
            A[index] ← a ;
            index ← index + 1 ;
        end
    end
```

**Funcionamento do Bucket Sort:** Seu primeiro passo é analisar todo o array para encontrar os valores mínimo (minValue) e máximo (maxValue). Com base nesse intervalo, ele cria uma lista de "baldes" (sub-listas vazias), geralmente em número igual à quantidade de elementos do array. Em seguida, o algoritmo percorre o array original elemento por elemento e, para cada um, utiliza uma fórmula matemática para mapear seu valor a um índice específico, distribuindo-o no balde correspondente. A fórmula essencialmente normaliza o valor do elemento dentro do intervalo [minValue, maxValue] para determinar em qual balde ele deve cair. Após todos os elementos serem distribuídos, o algoritmo passa por cada balde individualmente e o ordena, geralmente utilizando um método mais simples como o *Insertion Sort*, que é eficiente para listas pequenas. O passo final é percorrer a lista de baldes em ordem, do primeiro ao último, e concatenar todos os elementos de volta no array original, que agora estará completamente ordenado.

*C.Pigeonhole Sort*

**Algorithm 3:** Cocktail Sort

**Input:** Array $A$ de $n$ elementos
**Output:** Array $A$ ordenado
**Function** `CocktailSort(A)`**:**
  $n \leftarrow$ tamanho de $A$ ;
  **if** $n \leq 1$ **then**
    **return**;
  **end**
  $incio \leftarrow 0$ ;
  $fim \leftarrow n - 1$ ;
  $trocado \leftarrow$ **verdadeiro** ;
  **while** $trocado$ **do**
    $trocado \leftarrow$ **falso** ;
    **for** $i \leftarrow incio$ **to** $fim - 1$ **do**
      **if** $A[i] > A[i + 1]$ **then**
        Trocar $A[i]$ com $A[i + 1]$ ;
        $trocado \leftarrow$ **verdadeiro** ;
      **end**
    **end**
    **if** $\neg trocado$ **then**
      **break** ;
    **end**
    $fim \leftarrow fim - 1$ ;
    **for** $i \leftarrow fim\ incio + 1$ **do**
      **if** $A[i - 1] > A[i]$ **then**
        Trocar $A[i - 1]$ com $A[i]$ ;
        $trocado \leftarrow$ **verdadeiro** ;
      **end**
    **end**
    $incio \leftarrow incio + 1$ ;
  **end**

**Funcionamento do Pigeonhole Sort:** Primeiramente, o algoritmo determina o intervalo de valores presentes no array encontrando o valor mínimo (min) e o máximo (max). Com base nisso, ele calcula o tamanho do intervalo (range = max - min + 1) e cria uma lista de "escaninhos" (holes), que são listas vazias, uma para cada valor possível dentro do intervalo. O próximo passo é percorrer o array de entrada e colocar cada elemento em seu escaninho correspondente. O índice do escaninho para um elemento a é calculado de forma direta como a - min. Assim, o menor valor (min) vai para o escaninho 0, min + 1 vai para o escaninho 1, e assim por diante. Como cada escaninho é uma lista, o método lida naturalmente com valores duplicados. Por fim, o algoritmo reconstrói o array original percorrendo a lista de escaninhos em sequência, do índice 0 até o final, e adicionando todos os elementos de cada escaninho de volta ao array original. O resultado é um array perfeitamente ordenado.

## IV. Modelos de Aplicação

A escolha do algoritmo de ordenação mais adequado para um problema específico depende de diversos fatores, como o tamanho do conjunto de dados, a distribuição e o domínio dos valores, os requisitos de desempenho (tempo e espaço) e o ambiente de execução (memória disponível, linguagem de programação, paralelismo, etc).

### *A. Cocktail Sort*

O *Cocktail Sort*, também conhecido como *Bidirectional Bubble Sort*, é uma variação do *Bubble Sort* que se diferencia por percorrer o vetor em ambas as direções alternadamente. Essa varredura bidirecional permite que elementos pequenos "subam" rapidamente para o início e elementos grandes "desçam" rapidamente para o final, melhorando levemente o desempenho em listas parcialmente ordenadas.

Contudo, assim como o *Bubble Sort*, ele ainda possui complexidade $O(n^2)$ no pior e no caso médio, o que o torna altamente ineficiente em listas grandes. Em benchmarks comparativos, o *Cocktail Sort* pode ser de 8 a 13 vezes mais lento que algoritmos lineares ou baseados em divisão como *Quick Sort* ou *Merge Sort*.

**Aplicações práticas**:

- Ensino de algoritmos de ordenação devido à sua clareza e facilidade de implementação.
- Visualização de algoritmos em ferramentas educacionais.
- Pequenos conjuntos de dados, como listas com menos de 100 elementos, onde o desempenho não é crítico.

**Exemplo**: ordenação de uma pequena lista de tarefas por prioridade em um software didático ou protótipo visual.

**Limitações**:

- Pouco escalável para grandes volumes.
- Ineficiente em hardware de baixa capacidade se os dados forem extensos.
- Não se adapta bem à paralelização.

### *B. Bucket Sort*

O *Bucket Sort* é um algoritmo eficiente para ordenação de números reais ou inteiros com distribuição aproximadamente uniforme. O algoritmo divide o intervalo de valores em múltiplos baldes (*buckets*) e distribui os elementos entre eles. Cada balde é então ordenado de forma independente, e os baldes são finalmente concatenados.

A complexidade média do algoritmo é $O(n+k)$, sendo $k$ o número de baldes, e assume desempenho quase linear quando os dados estão bem distribuídos. Esse tipo de ordenação se destaca por não depender de comparações diretas entre todos os pares de elementos.

**Aplicações práticas**:

- Ordenação de valores contínuos em domínios limitados, como médias escolares (0 a 10), temperaturas, tempos de resposta.
- Processamento de dados de sensores com ruído previsível.
- Visualização de histogramas e agrupamentos dinâmicos.

**Exemplo**: agrupamento e ordenação de notas de alunos em uma plataforma educacional, onde as notas vão de 0 a 100 e são distribuídas de forma relativamente uniforme.

**Limitações**:

- Desempenho degrada-se se os dados não forem uniformemente distribuídos.

- Exige conhecimento prévio do intervalo de valores.
- Complexidade de implementação aumenta em ambientes paralelos ou com dados em tempo real.

### C. *Pigeonhole Sort*

O *Pigeonhole Sort* é um algoritmo que se baseia no princípio das casas de pombo: se houver mais itens do que locais para armazená-los, pelo menos um local terá mais de um item. Este algoritmo é aplicável quando os elementos a serem ordenados são inteiros dentro de um intervalo pequeno conhecido.

Sua complexidade é linear, $O(n + r)$, onde $r$ representa o tamanho do intervalo de valores possíveis. Ele é especialmente vantajoso quando o número de elementos é próximo ao número de valores distintos possíveis por exemplo, ao ordenar dados com baixa variabilidade e alta densidade.

**Aplicações práticas**:

- Ordenação de faixas etárias (por exemplo, de 0 a 120 anos).
- Classificação de categorias ou níveis (nível de acesso, avaliações inteiras, contagens).
- Agrupamento de valores discretos com poucos elementos distintos.

**Exemplo**: em um sistema de votação onde os votos são inteiros de 1 a 5 (avaliação por estrelas), o *Pigeonhole Sort* pode ordenar rapidamente uma grande quantidade de avaliações.

**Limitações**:

- Consumo excessivo de memória quando o intervalo de valores é muito grande e os dados são esparsos.
- Ineficiente para dados com alta variação e baixa densidade (ex: IDs aleatórios).
- Requer que todos os valores estejam previamente no intervalo definido.

### *Comparativo Geral*

TABLE I: Comparação entre algoritmos de ordenação

| Algoritmo | Complexidade | Memória | Melhor uso | Limitações |
|---|---|---|---|---|
| Cocktail | $O(n^2)$ | Baixa | Lista pequena | Fraco c/ muitos dados |
| Bucket | $O(n + k)$ | Média | Dados contínuos | Precisa uniformidade |
| Pigeonhole | $O(n + r)$ | Alta | Inteiros próximos | Usa muita memória |

## V. Resultados

A análise dos algoritmos *Bucket Sort*, *Cocktail Sort* e *Pigeonhole Sort* foi levada demodo a permitir uma avaliação detalhada de seu desempenho em diferentes condições. Os testes foram realizados em vetores desordenados, com variados tamanhos de entrada (100, 1.000, 10.000, 100.000, 1.000.000). Adicionalmente,considerou-se o desempenho nos códigos, utilizando lista, fila e pilha, em linguagens C, C++, Java, Python e Ruby. Essa abordagem multifacetada permitiu analisar como diferentes fatores influenciam o tempo utilizado de cada algoritmo e avaliar seu desempenho em diversos contextos, levando em conta também a memória utilizada na execução dos arquivos.

Para a testagem dos algoritmos, foi utilizada uma máquina com o sistema operacional GNU/Linux Debian com ambiente gráfico KDE, equipada com processador AMD Ryzen 7 5700X, 8 núcleos físicos e 16 threads, 32 GB de memória RAM e armazenamento em SSD NVMe.

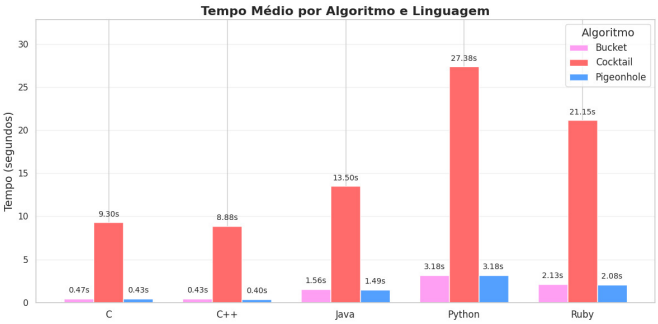


Fig. 1: Tempo Médio por Algoritmo e Linguagem

Na imagem do gráfico 1, temos uma análise geral do tempo médio do algoritmo por linguagem, levando em conta cada algoritmo separadamente. Mormente, nota-se *Cocktail Sort* sendo o mais demorado e Python a executar.

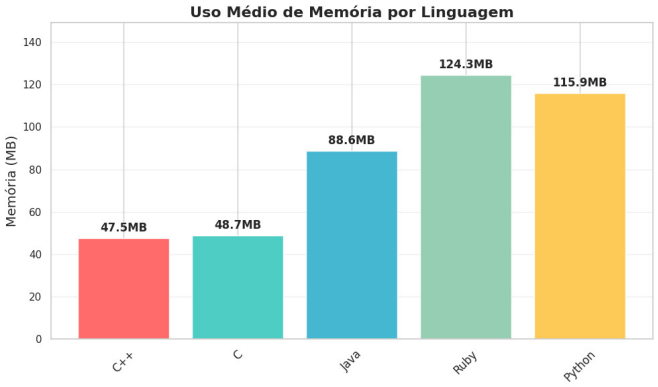


Fig. 2: Gráfico de Memória vs Linguagem

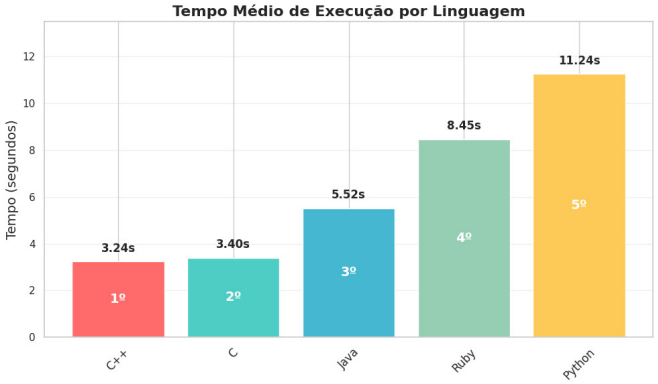


Fig. 3: Gráfico de Tempo vs Linguagem

Na figura 2, destaca-se o maior uso de memória da linguagem Ruby, e, nota-se na figura 3, como Python levou mais tempo para executar, como foi ressaltado nos gráficos anteriores. Além disso, destaca-se como C++ mostrou-se a linguagem mais rápida e com menor uso de memória.

### A. Resultados em C

A Figura 4 apresenta os resultados obtidos na linguagem C. O *Cocktail Sort* demonstrou o pior desempenho em tempo e memória, especialmente quando implementado com a estrutura de pilha-vetor. Da mesma forma, o *Bucket Sort* com lista-ponteiro também apresentou um consumo elevado de memória, o que pode ser explicado pelo uso de estruturas encadeadas, que adicionam sobrecarga ao gerenciamento dos dados. No geral, o C mostrou-se eficiente, mas sensível às estruturas de dados utilizadas.

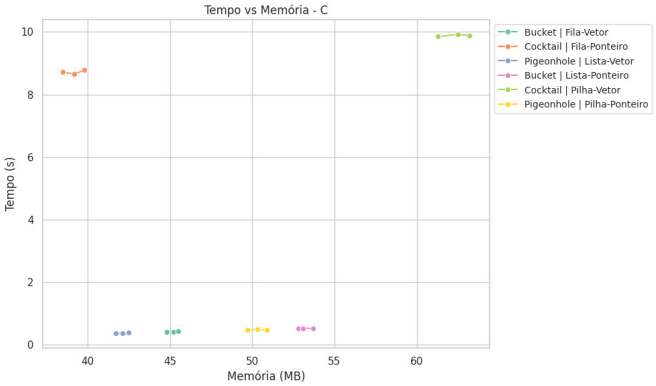


Fig. 4: Gráfico de TempoxMemória em C

### B. Resultados em C++

Na Figura 5, vemos os resultados obtidos com a linguagem C++. Assim como em C, o *Cocktail Sort* apresentou maior uso de memória em comparação aos demais algoritmos. No entanto, houve uma leve melhora nos tempos de execução em relação ao C, possivelmente devido às otimizações realizadas pelo compilador da linguagem C++. Essa diferença, embora pequena, pode indicar uma vantagem dalinguagem em contextos que exigem melhor desempenho computacional.

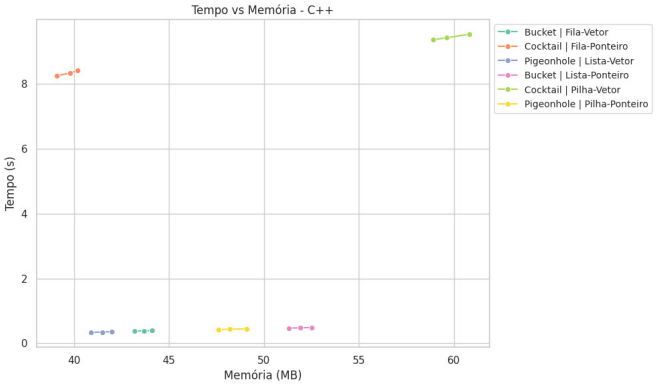


Fig. 5: Gráfico de Tempo vs Memória em C++

### C. Resultados em Java

Os algoritmos de ordenação foram testados também em Java. Uma linguagem multiplataforma, utilizada de sistemas corporativos e aplicativos móveis até jogos. Muito conhecida por portabilidade e desempenho é uma das linguagens de programação mais utilizadas no mundo. Assim, os gráficos abaixo demonstram o desempenho dos métodos de ordenação estudados.

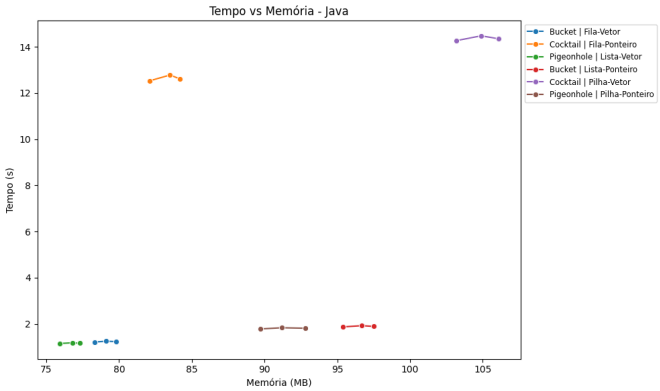


Fig. 6: Gráfico de Tempo vs Memória em Java

Como mostrado no gráfico 6, os algoritmos *Bucket Sort* e *Pigeonhole Sort* tiveram um desempenho bem melhor em relação ao tempo do que o desempenho mostrado do *Cocktail Sort*, tanto com o uso de vetores quanto com a utilização de ponteiros. No quesito memória, assim como no tempo, foi uma linguagem mediana em comparação com as outras.

### D. Resultados em Python

A Figura 7 ilustra o desempenho dos algoritmos em Python. A linguagem apresentou os piores tempos de execução entre todas, sendo aproximadamente três vezes mais lenta que C++ em média. Esse resultado já era esperado, considerando que Python é uma linguagem interpretada, com uma camada adicional de abstração que afeta diretamente sua performance. Além disso, o uso de memória também foi elevado, refletindo o custo de estruturas internas mais complexas e do gerenciamento automático de memória.

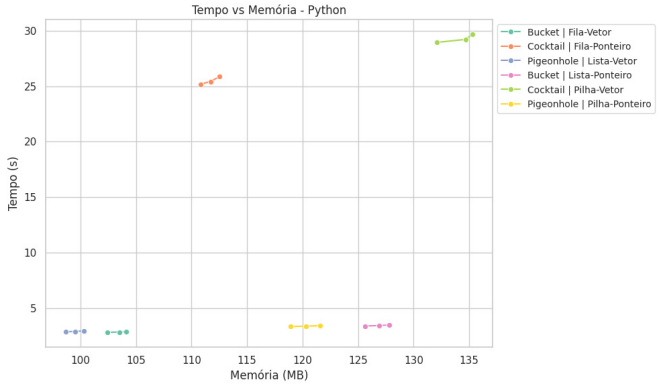


Fig. 7: Gráfico de Tempo vs Memória em Python

### E. Resultados em Ruby

Por fim, a Figura 8 mostra os resultados obtidos com a linguagem Ruby. O desempenho ficou em uma posição intermediária: melhor que o de Python, mas ainda inferior ao de C e C++. O tempo de execução foi aceitável, mas o consumo de memória foi relativamente alto, indicando que, embora Ruby seja mais rápido que Python em alguns casos, seu uso ainda deve ser ponderado em aplicações que exigem alta eficiência.

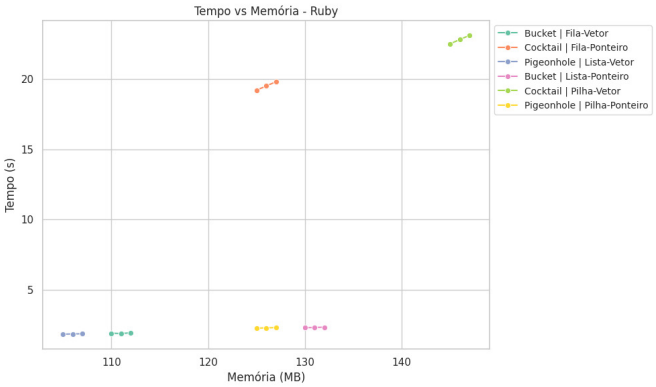


Fig. 8: Gráfico de Tempo vs Memória em Ruby

TABLE II: Nº Usuários por Segundo – C

| Estrutura | Bucket Sort | Cocktail Sort | Pigeonhole Sort |
|---|---|---|---|
| Lista | 172340 | 17445 | 188953 |
| Fila | 172340 | 17453 | 190588 |
| Pilha | 168750 | 17364 | 188506 |

TABLE III: Nº Usuários por Segundo – C++

| Estrutura | Bucket Sort | Cocktail Sort | Pigeonhole Sort |
|---|---|---|---|
| Lista | 189535 | 18409 | 207792 |
| Fila | 184091 | 18277 | 205063 |
| Pilha | 180000 | 18066 | 197561 |

TABLE IV: Nº Usuários por Segundo – Java

| Estrutura | Bucket Sort | Cocktail Sort | Pigeonhole Sort |
|---|---|---|---|
| Lista | 101887 | 11886 | 107285 |
| Fila | 105195 | 12089 | 110345 |
| Pilha | 103846 | 12019 | 108725 |

TABLE V: Nº Usuários por Segundo – Python

| Estrutura | Bucket Sort | Cocktail Sort | Pigeonhole Sort |
|---|---|---|---|
| Lista | 50311 | 5836 | 50311 |
| Fila | 51592 | 5987 | 51592 |
| Pilha | 50943 | 5934 | 50943 |

TABLE VI: Nº Usuários por Segundo – Ruby

| Estrutura | Bucket Sort | Cocktail Sort | Pigeonhole Sort |
|---|---|---|---|
| Lista | 50000 | 5615 | 50000 |
| Fila | 50625 | 5684 | 50625 |
| Pilha | 50467 | 5658 | 50467 |

Percebe-se então que de acordo com o que foi mostrado nos gráficos e nas tabelas, é possível demonstrar que cada linguagem reage de uma maneira única dependendo da estrutura que foi usada, como lista, fila e pilha, e ainda mais variações surgem quando são utilizados os ponteiros ou os vetores, cada um reagindo de uma maneira única. Cada método de ordenação seguindo seu próprio tempo e desenvolvimento, como por exemplo a implementação do *Cocktail Sort* em Java, que diminui drasticamente o número de usuários por segundo quando comparado com os outros algoritmos na mesma linguagem.

## VI. Conclusão

Os algoritmos *Cocktail Sort*, *Bucket Sort* e *Pigeonhole Sort* ilustram abordagens alternativas à ordenação clássica, cada um com características que os tornam mais ou menos adequados para diferentes cenários.

O *Cocktail Sort*, apesar de simples, oferece um ganho modesto sobre o *Bubble Sort* em vetores parcialmente ordenados, sendo útil em contextos didáticos ou com pequenas entradas. O *Bucket Sort* demonstrou excelente desempenho com dados distribuídos de forma uniforme, mostrando-se competitivo com algoritmos mais avançados. Já o *Pigeonhole Sort* é extremamente eficiente quando o intervalo de valores é pequeno e bem definido, mas sua aplicação é limitada em conjuntos grandes com grande dispersão de dados.

Os testes com o arquivo `ratings.csv` evidenciaram essas características, mostrando que o conhecimento profundo sobre o domínio dos dados é essencial para escolher o algoritmo de ordenação mais apropriado. Como linha futura de pesquisa, propõe-se o estudo de versões paralelas ou híbridas desses algoritmos, bem como sua aplicação em domínios específicos como processamento de imagens, redes neurais e análise em tempo real.

## VII. Algoritmos

Os códigos-fonte implementando os algoritmos estudados neste trabalho, nas linguagens **C**, **C++**, **Java**, **Python** e **Ruby**, estão disponíveis no seguinte repositório público:

https://github.com/KairoHenrique/Ordenacoes_Bidirecionais_e_Distribuidas

## REFERENCES


[1] Kaggle.com, "Movielens 25m dataset." https://www.kaggle.com/datasets/garymk/movielens-25m-dataset. Acesso em: 27 jun. 2025.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 3rd ed., 2009.

[3] D. E. Knuth, "The art of computer programming, volume 3: Sorting and searching," *Addison-Wesley*, 1998.

[4] GeeksforGeeks, "Bucket sort | set 1 (introduction) - geeksforgeeks." https://www.geeksforgeeks.org/bucket-sort-2/, 2023. Acesso em: 23 jun. 2025.

[5] Gautam007, "Bucket sort: Efficient sorting for certain distributions." https://medium.com/@gautam007/bucket-sort-efficient-sorting-for-certain-distributions-3d709ef824bd, 2023. Acesso em: 22 jun. 2025.

[6] Quora, "What types of applications can benefit from bucket sorting algorithms." https://www.quora.com/What-types-of-applications-can-benefit-from-bucket-sorting-algorithms, 2023. Acesso em: 25 jun. 2025.

[7] S. Overflow, "When should i choose bucket sort over other sorting algorithms?." https://stackoverflow.com/questions/31633391/when-should-i-choose-bucket-sort-over-other-sorting-algorithms, 2023. Acesso em: 22 jun. 2025.

[8] GeeksforGeeks, "Cocktail sort – dsa." https://www.geeksforgeeks.org/dsa/cocktail-sort/, 2023. Acesso em: 23 jun. 2025.

[9] P. Kanoongo, "Cocktail sort." https://palakkanoongo2k1.medium.com/cocktail-sort-bdd2f4df51ed, 2023. Acesso em: 25 jun. 2025.

[10] G. 7, "Bubble and cocktail sort." https://www.scribd.com/document/562975437/GROUP7-BUBBLE-AND-COCKTAIL-SORT, 2023. Acesso em: 25 jun. 2025.

[11] Siberoloji, "Cocktail sort in computer algorithms." https://www.siberoloji.com/cocktail-sort-in-computer-algorithms/, 2023. Acesso em: 25 jun. 2025.

[12] W. contributors, "Cocktail shaker sort." https://en.wikipedia.org/wiki/Cocktail_shaker_sort, 2023. Acesso em: 25 jun. 2025.

[13] GeeksforGeeks, "Cocktail sort - geeksforgeeks." https://www.geeksforgeeks.org/cocktail-sort/, 2023. Acesso em: 25 jun. 2025.

[14] GeeksforGeeks, "Pigeonhole sort – dsa." https://www.geeksforgeeks.org/dsa/pigeonhole-sort/, 2023. Acesso em: 23 jun. 2025.

[15] GeeksforGeeks, "Python program for pigeonhole sort." https://www.geeksforgeeks.org/python/python-program-for-pigeonhole-sort/, 2023. Acesso em: 24 jun. 2025.

[16] GeeksforGeeks, "Sorting algorithms in python." https://www.geeksforgeeks.org/sorting-algorithms-in-python/, 2023. Acesso em: 24 jun. 2025.

[17] ICMC-USP, "Slides sobre ordenação – scc0601 (2011)." http://wiki.icmc.usp.br/images/0/0a/SCC0601-2oSem2011-Lucas-Slides13.pdf, 2011. Acesso em: 24 jun. 2025.

[18] T. C. L. B. Game, "Performance benchmarks - c++, java, python." https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Acesso em: 25 jun. 2025.

[19] GeeksforGeeks, "Pigeonhole sort – geeksforgeeks." https://www.geeksforgeeks.org/pigeonhole-sort/, 2023. Acesso em: 25 jun. 2025.

[20] Siberoloji, "Pigeonhole sort in computer algorithms." https://www.siberoloji.com/pigeonhole-sort-in-computer-algorithms/, 2023. Acesso em: 22 jun. 2025.

[21] GeeksforGeeks, "In which case is pigeonhole sort best? – geeksforgeeks," n.d. Acesso em: 26 jun. 2025.

[22] P. Singla, "Lecture 14: Sorting (col106, iit delhi)," 2022. Acesso em: 27 jun. 2025.

[23] Educative, "Comparison of linear time sorting algorithms," n.d. Acesso em: 26 jun. 2025.

[24] D. Holanda, I. Moreira, K. Henrique, and T. Luisa, "Ordenações bidirecionais e distribuídas." https://github.com/KairoHenrique/Ordenacoes_Bidirecionais_e_Distribuidas, 2025. Acesso em: 29 jun. 2025.