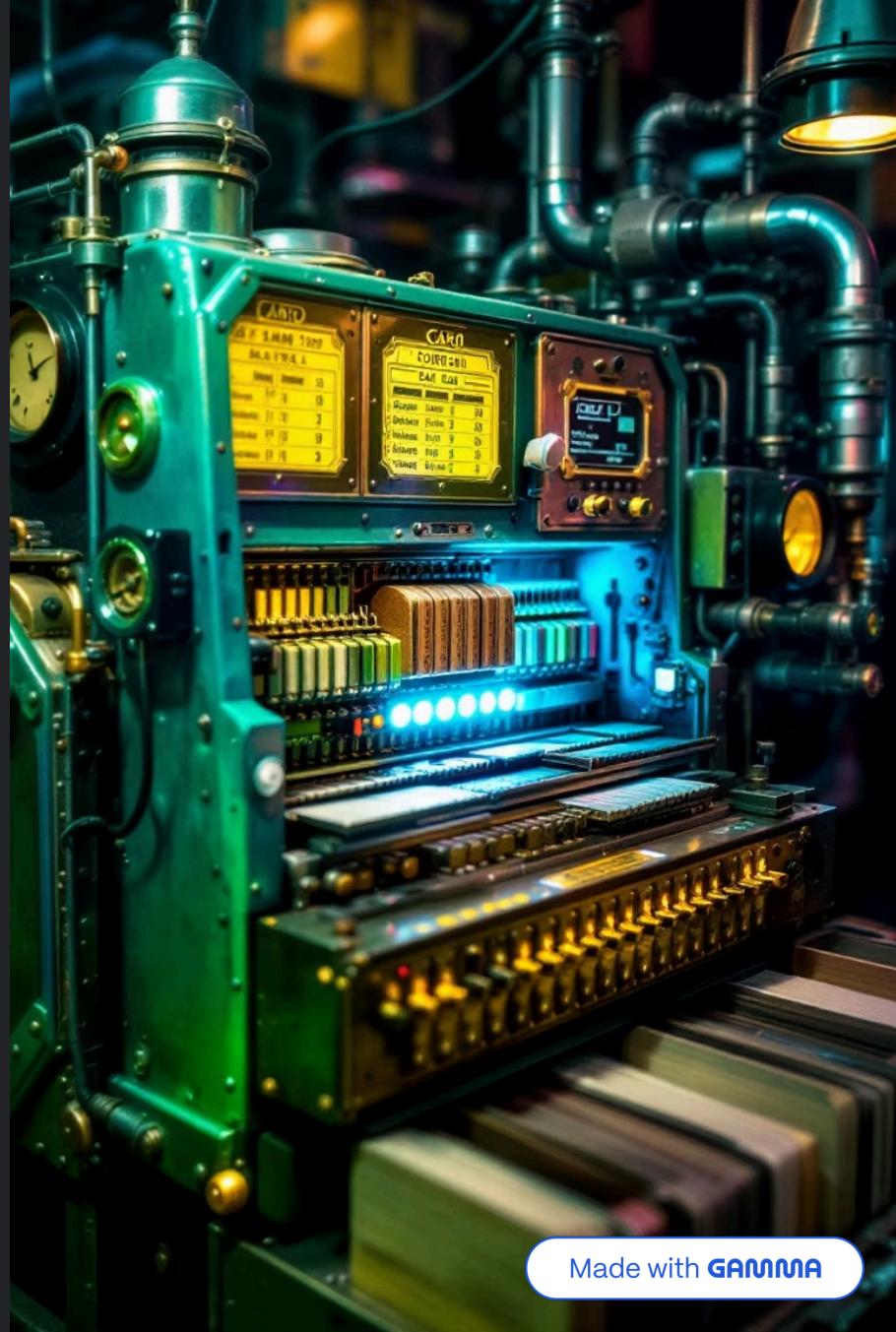
Algoritmos de Ordenação Não Comparativos: Fundamentos e História

Os algoritmos de ordenação não comparativos representam uma fascinante evolução tecnológica que conecta as máquinas de tabulação mecânicas do século XIX aos modernos sistemas computacionais do século XXI. **Counting Sort e Radix Sort são algoritmos únicos que conseguem alcançar complexidade linear O(n) sob condições específicas**, rompendo o limite inferior teórico de Ω(n log n) que governa os algoritmos comparativos tradicionais.

A jornada desses algoritmos começa em 1887 com Herman Hollerith e culmina nas descobertas contemporâneas de inteligência artificial da Google DeepMind em 2023, demonstrando sua relevância contínua na ciência da computação.

B J **2 colaboradores**



A Revolução Mecânica de Herman Hollerith (1887-1920)

1884

Hollerith registra a patente "Art of Compiling Statistics" (Patente US 395.782), criando um sistema eletromecânico revolucionário.

1890

Sistema de Hollerith é usado no Censo Americano, reduzindo o tempo de processamento de mais de 8 anos para menos de 1 ano.

1896

Fundação da Tabulating Machine Company, que eventualmente se transformaria na IBM através de fusões em 1911 e 1924.

1970

As máquinas de cartão perfurado da IBM dominam o processamento de dados empresariais até esta década.

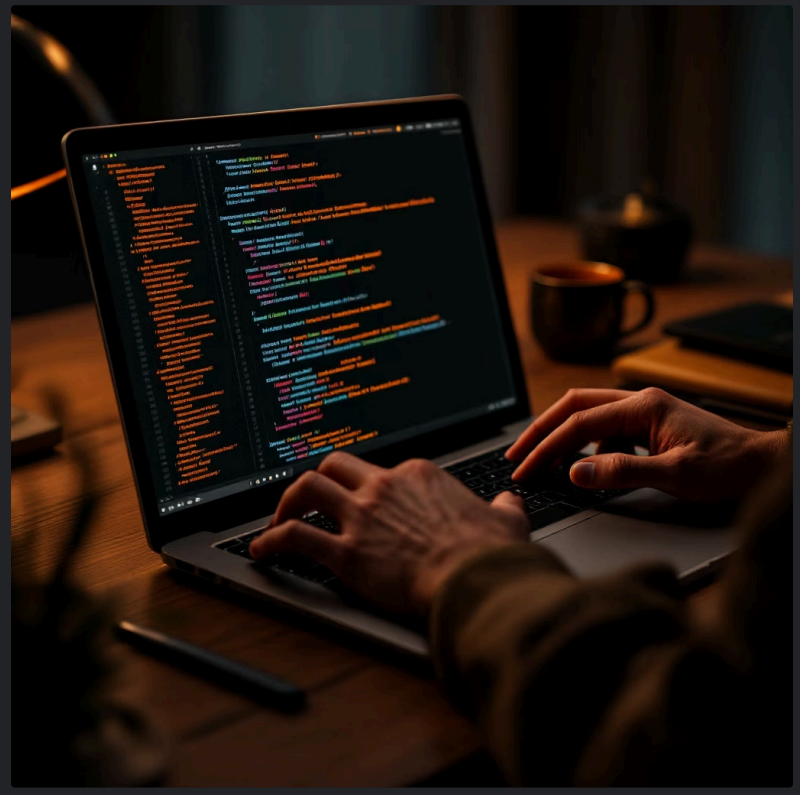
A origem do Radix Sort remonta a **Herman Hollerith (1860-1929)**, que desenvolveu o primeiro sistema de ordenação não comparativa para as máquinas tabuladoras do Censo Americano de 1890. O sistema utilizava **pinos com molas, poços de mercúrio e circuitos elétricos** para detectar perfurações nos cartões, classificando-os automaticamente em 13 compartimentos baseados nos padrões de furos.

História do Radix Sort

A história do Radix Sort tem suas raízes no final do século XIX, com as primeiras ideias e implementações surgindo em um contexto muito distinto dos computadores eletrônicos modernos.

Seu conceito fundamental foi explorado por Herman Hollerith, inventor do cartão perfurado e fundador da Tabulating Machine Company (precursora da IBM). Em 1890, a máquina de tabulação de Hollerith foi crucial para processar o Censo dos Estados Unidos, onde utilizava cartões perfurados para ordenar dados com base na posição dos dígitos – um princípio que diretamente inspirou o funcionamento do Radix Sort.

O algoritmo foi significativamente modernizado nas décadas de 1950 e 1960, passando a ser aplicado em computadores para ordenar dados estruturados como números e códigos. Atualmente, ele funciona processando os dígitos de cada número, frequentemente utilizando algoritmos estáveis como o Counting Sort em cada etapa, o que lhe permite alcançar desempenho linear em certos cenários.



Funcionamento do Counting Sort

CONTAGEM ORDENADA: O SEGREDO DOS ALGORITMOS NÃO COMPARATIVOS

Vamos desvendar os passos fascinantes do Counting Sort:

1. Identifique o Dígito Decisivo

1

Tudo começa com a análise do dígito menos significativo de cada número.

2. Conte as Ocorrências

2

Criamos um vetor especial para contabilizar quantas vezes cada valor aparece.

3. Posicione com Precisão

3

Usando a contagem acumulada, determinamos a posição final de cada elemento.

4. Ordene com Maestria

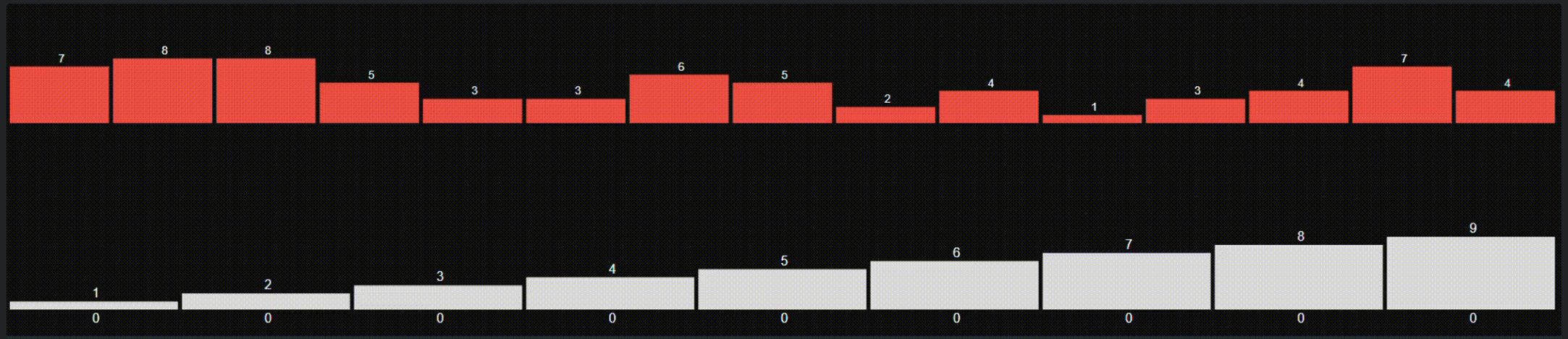
4

Finalmente, construímos o array final, com todos os elementos perfeitamente organizados.

PSEUDOCÓDIGO

```
Função CountingSort(lista)
    Criar vetor keys[9] e inicializa com zeros
    Criar vetor results (vazio)

    Para i de 0 até tamanho(lista) - 1:
            keys[lista[i] - 1]++

    Para k de 0 até 8:
        Enquanto keys[k] != 0:
            results.push_back(k + 1)
                keys[k]--

    Retornar results
Fim da função
```

Counting Sort: Visualização do Funcionamento




```
keys = [0,0,0,0,0,0,0,0,0]
for(i = 0 to length(list))
  keys[list[i]-1]++;
for(k = 0 to length(keys))
  while(keys[k] != 0)
    results.push(k+1)
    keys[k]--;
```

```
SHORT EXPLANATION
-----------------
1.   Initialize the key array with zeros.
2.   For every number in the list increase the respective index in the key array.
3.   Loop over all elements of the key array.
3.1. While the current element is not zero, add the index to the result
     and decrease the counter.
```

Radix Sort: Fucionamento

1 Seleciona o Dígito Menos Significativo

Começamos analisando o dígito menos importante de cada número.

2 Ordena com Counting Sort

Usamos o Counting Sort para reorganizar os elementos com base nesse dígito.

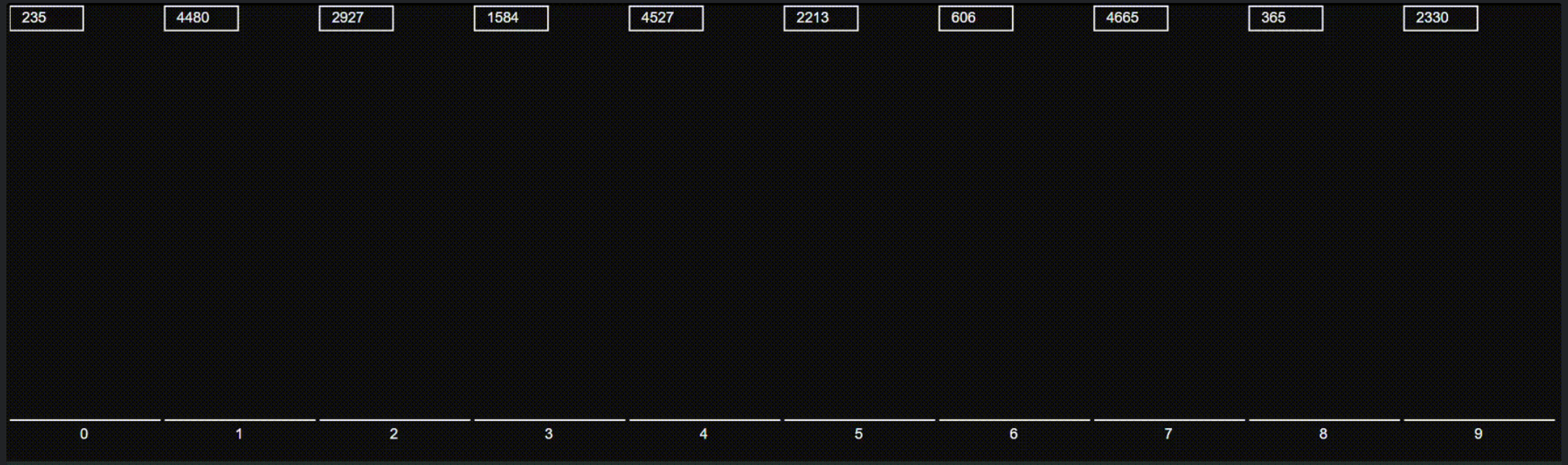
3 Repete para Próximos Dígitos

Continuamos o processo, avançando para os dígitos mais significativos, até que todo o array esteja perfeitamente ordenado.

PSEUDOCÓDIGO

```
Função RadixSort(lista)
    numDigitos ← número de dígitos do maior valor em lista

    Para i de 0 até numDigitos - 1:
        Criar vetor buckets com 10 listas vazias (índices de 0 a 9)

        Para j de 0 até tamanho(lista) - 1:
            num ← obter o dígito da posição i do número lista[j]
            Se num for válido:
                Inserir lista[j] em buckets[num]

        lista ← juntar todos os valores dos buckets na ordem de 0 a 9

    Retornar lista
Fim da função
```

Radix Sort: Visualização do Funcionamento




```
for i = 0 to numDigits(data.max)
  buckets = new Array(10)
  for x = 0 to length(data)
    num = getNum(data[j], i)
    if num !== undefined
      buckets[num].push(data[j])
  data = mergeBuckets()
```

```
| SHORT EXPLANATION |
1. Create ten empty arrays (the buckets which will later hold the numbers)
2. For each element in the list, move it into the correct bucket.
3. For each bucket (starting from 0), restore elements to the list.
4. Repeat steps 2 and 3 until all digits have been seen.
```

Relação Simbiótica entre Counting Sort e Radix Sort

Radix Sort

Decompõe o problema em múltiplas ordenações de menor escala por dígito

Counting Sort

Utilizado como sub-rotina para ordenar cada posição de dígito

Eficiência

Resolve o problema de escalabilidade quando k ›› n

Estabilidade

Preserva a ordem relativa de elementos com dígitos iguais

A **relação entre Counting Sort e Radix Sort é fundamental e simbiótica**. O Radix Sort utiliza o Counting Sort como sub-rotina para ordenar cada posição de dígito, aproveitando sua estabilidade para manter a ordem correta estabelecida pelas iterações anteriores.

Complexidade do Counting Sort

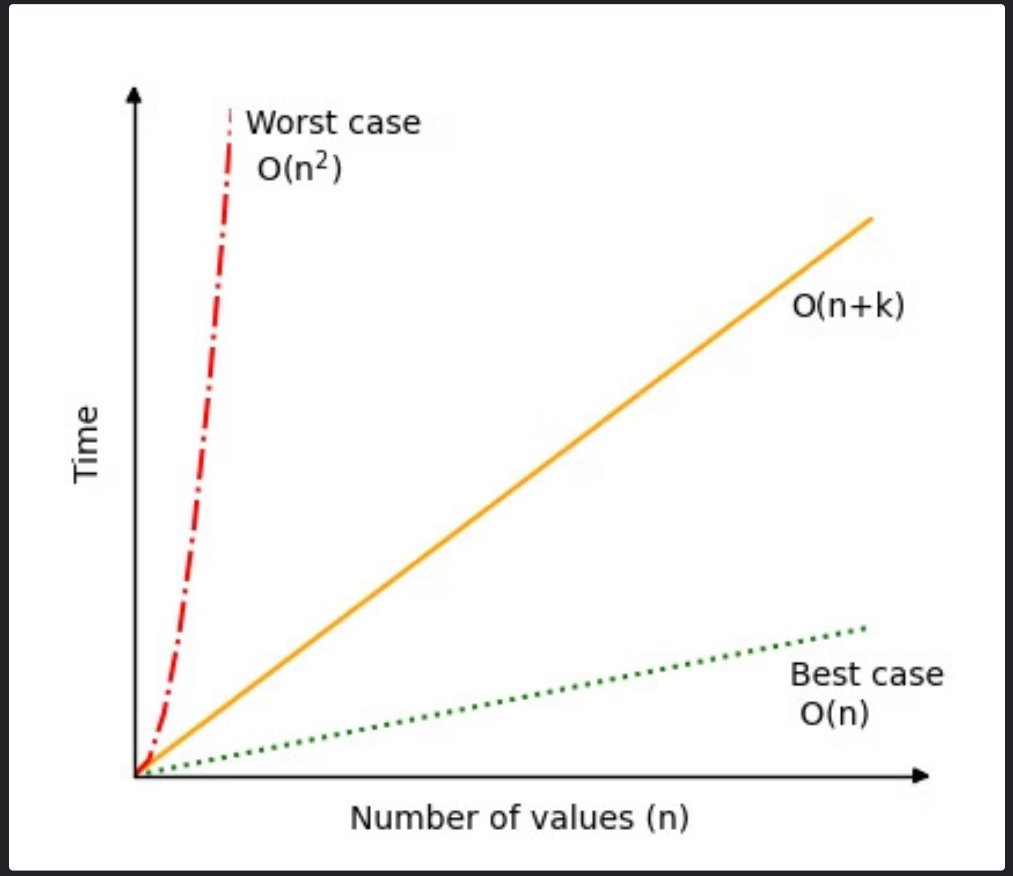
Elementos Chave e Análise de Tempo

A complexidade do Counting Sort depende de dois fatores: **n** (o número de elementos na lista) e **k** (a amplitude do intervalo dos números, ou seja, o maior valor menos o menor valor mais um).

O algoritmo executa as seguintes etapas:

- **Inicialização do array de contagem:** O(k)
- **Contagem de frequências dos elementos:** O(n)
- **Modificação do array de contagem para posições:** O(k)
- **Construção do array de saída:** O(n)

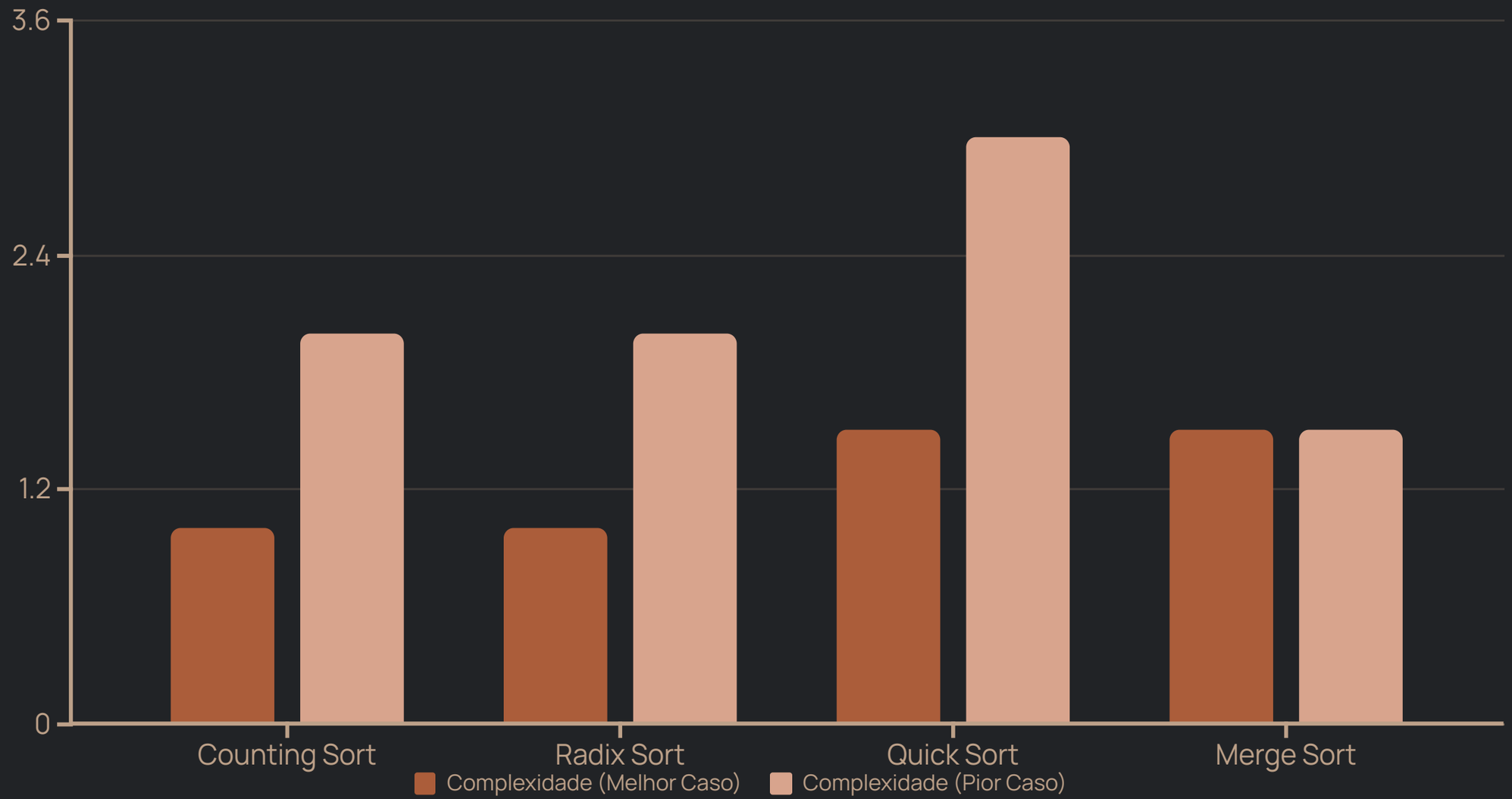
Somando as complexidades de cada passo, obtemos uma complexidade total de **O(n + k)**.



Vantagem e Condições

Esta complexidade linear (em relação a n e k) torna o Counting Sort **extremamente eficiente** quando o intervalo dos valores (k) não é significativamente maior que o número de elementos (n). Se k for muito grande, o algoritmo pode se tornar ineficiente devido ao uso de memória e tempo para inicializar o array de contagem.

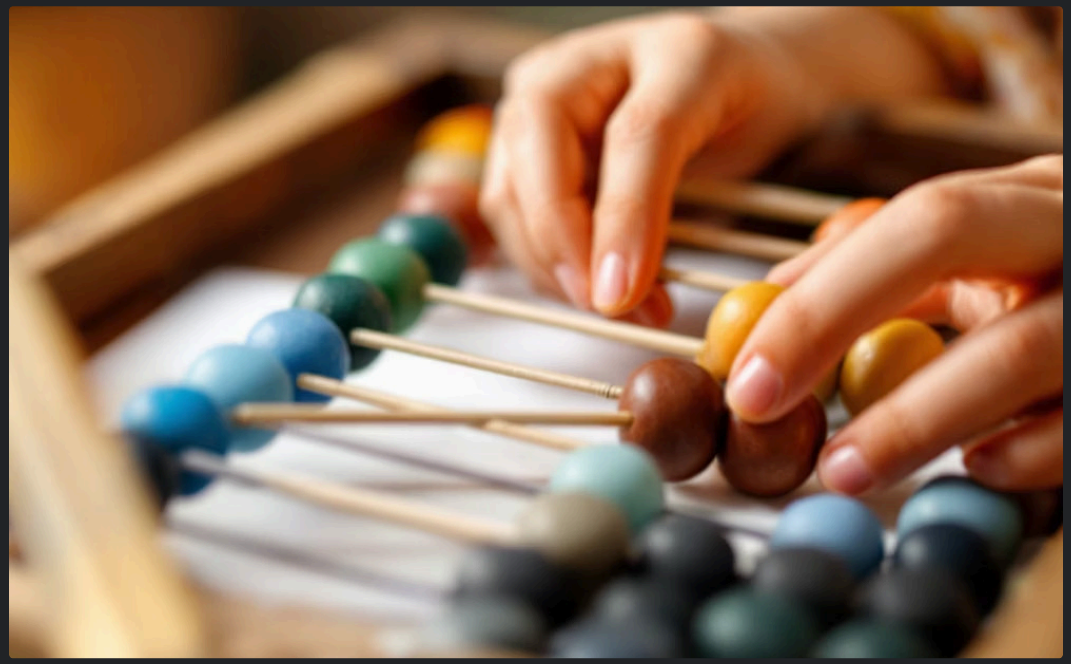
Complexidade do Counting Sort e Radix Sort



O **Counting Sort alcança complexidade O(n + k)**, onde n é o número de elementos e k é o intervalo de valores. O algoritmo atinge O(n) quando k = O(n) - ou seja, quando o intervalo de valores é proporcional ao número de elementos.

O **Radix Sort apresenta complexidade O(d × (n + b))**, onde d é o número de dígitos, n é o número de elementos, e b é a base utilizada. O algoritmo alcança O(n) quando d é constante e b = O(n). Para inteiros de w bits usando base b, o número de dígitos é $d = \lceil w/\log_2(b) \rceil$.

Uso: Quando usar cada um?



Counting Sort

Ideal quando os dados são inteiros com um **intervalo de valores relativamente pequeno (0 a k)**. É extremamente rápido para conjuntos de dados com **valores frequentemente repetidos.**

Radix Sort

Perfeito para ordenar grandes volumes de **inteiros com um número fixo de dígitos**. É particularmente eficiente para números longos, como **CPFs, CNPJs, números de telefone ou chaves de banco**.

Desvantagens/ Limitações do Counting Sort

Apesar de eficientes sob certas condições, os algoritmos **Counting Sort** e **Radix Sort** apresentam limitações importantes que afetam sua aplicabilidade em cenários gerais.

🧮 Counting Sort

- **Necessita conhecimento prévio do maior valor (k)**
  → Não é prático quando **k é muito grande**.
- **Uso excessivo de memória**
  → Utiliza um vetor auxiliar de tamanho proporcional a `k`.
- **Limitado a inteiros ou chaves discretas**
  → Não funciona bem com dados complexos, como strings ou objetos customizados.
- **Não é in-place**
  → Precisa de estrutura auxiliar para armazenar o resultado ordenado.

Aplicações e Impacto Prático Contemporâneo

Processamento de grandes volumes de dados

Ideais para ordenar grandes quantidades de dados numéricos em sistemas como bancos, e-commerces e redes sociais, quando os valores seguem uma faixa limitada.

Sistemas de Tempo Real

Excelente desempenho em contextos que exigem resposta rápida, como análise de logs, pacotes de rede e sistemas embarcados, graças à sua complexidade linear.

Computação gráfica e jogos

Usados para ordenar objetos por profundidade ou cor de forma eficiente em engines gráficas, com desempenho superior para faixas discretas de valores.

Machine Learning e Big Data

Aplicados no pré-processamento de dados categóricos ou inteiros, otimizando a etapa de organização e discretização em grandes volumes de dados.

A relevância contínua destes algoritmos é evidenciada pelas descobertas recentes da IA, mostrando que mesmo algoritmos considerados "resolvidos" podem ser otimizados através de novas abordagens computacionais.

Metodologia dos Testes

Para garantir a precisão e a reprodutibilidade dos experimentos de performance, os testes com o algoritmo **Counting Sort** seguiram a metodologia descrita abaixo:

Dados de Entrada

- Conjuntos de dados gerados aleatoriamente.
- Variedade de tamanhos de 100 a 1000000
- Distribuição uniforme e não uniforme dos valores.

Ambiente de Execução

- Processador: Ryzen 5 5500U
- Memória RAM: 8GB
- Sistema Operacional: **Linux 24.04 LTS**
- Linguagens de Programação: C++, C, Go, Python, Java

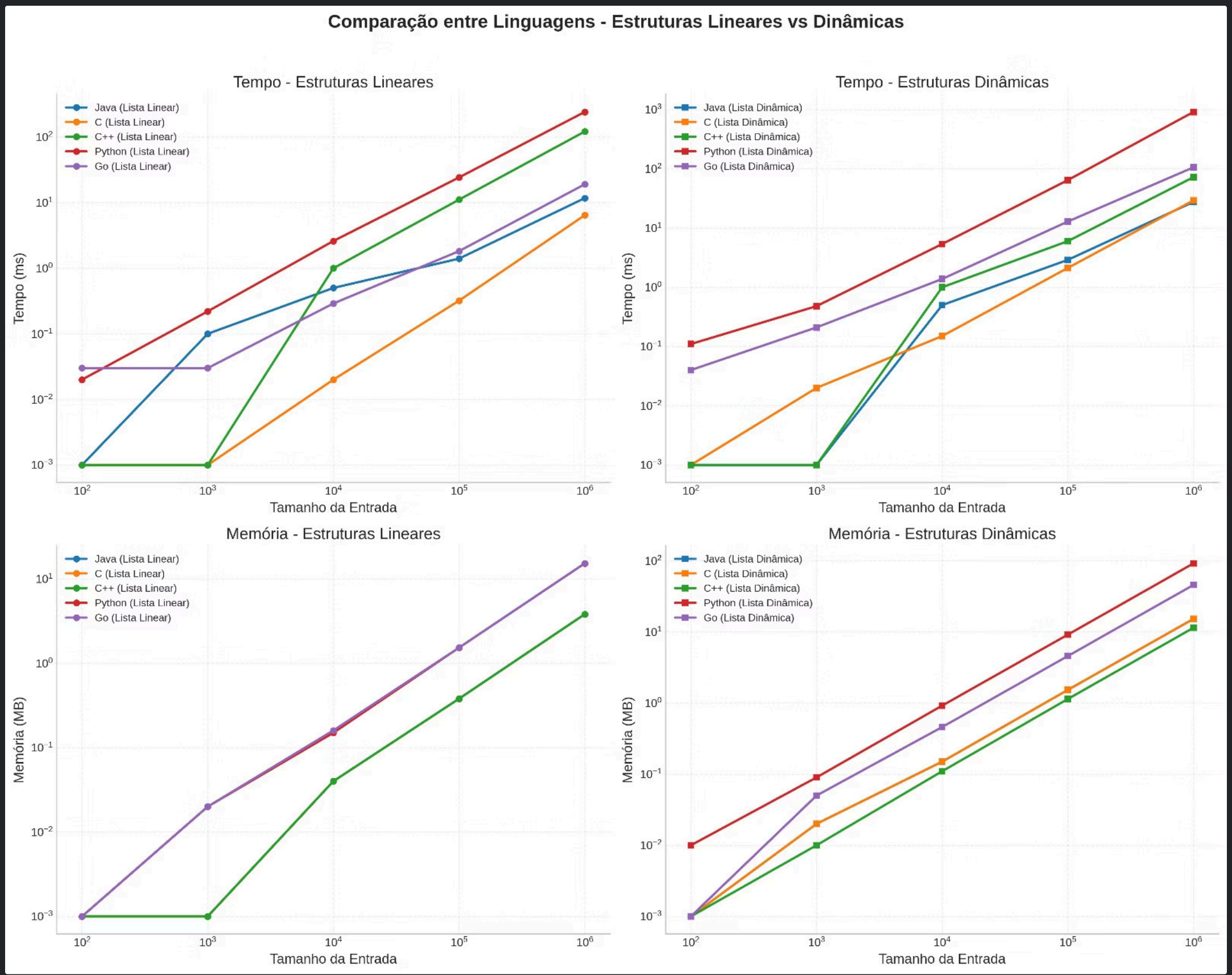
Execução dos Testes

- Cada teste executado múltiplas vezes
- Média dos tempos de execução para mitigar ruídos.
- Início "frio" para cada execução (sem cache ou otimizações prévias).
- O algoritmo avaliado foi exclusivamente o **Counting Sort**.

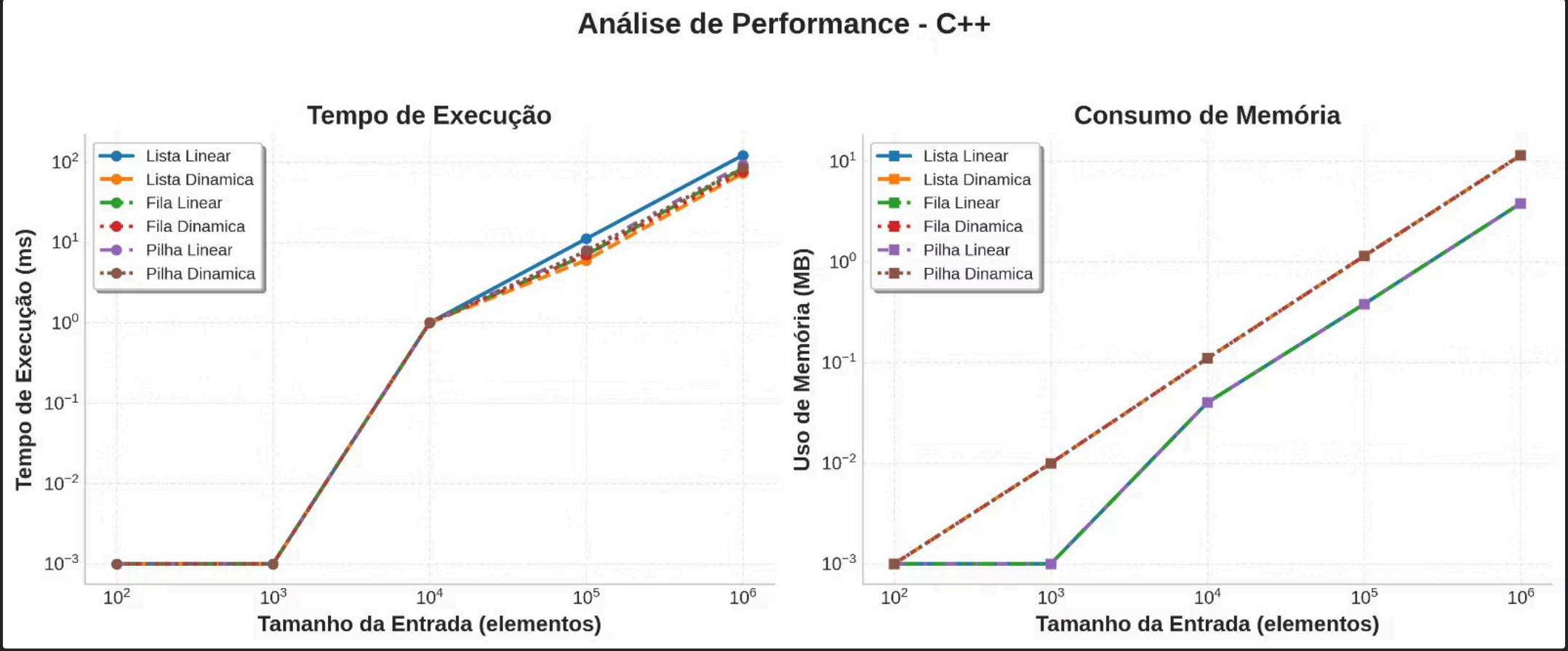
Critérios Avaliados

- Tempo de Execução (em milissegundos).
- Uso de Memória (em KB ou MB).
- Consistência dos resultados entre as repetições.
- Impacto do tamanho de N e K na performance.

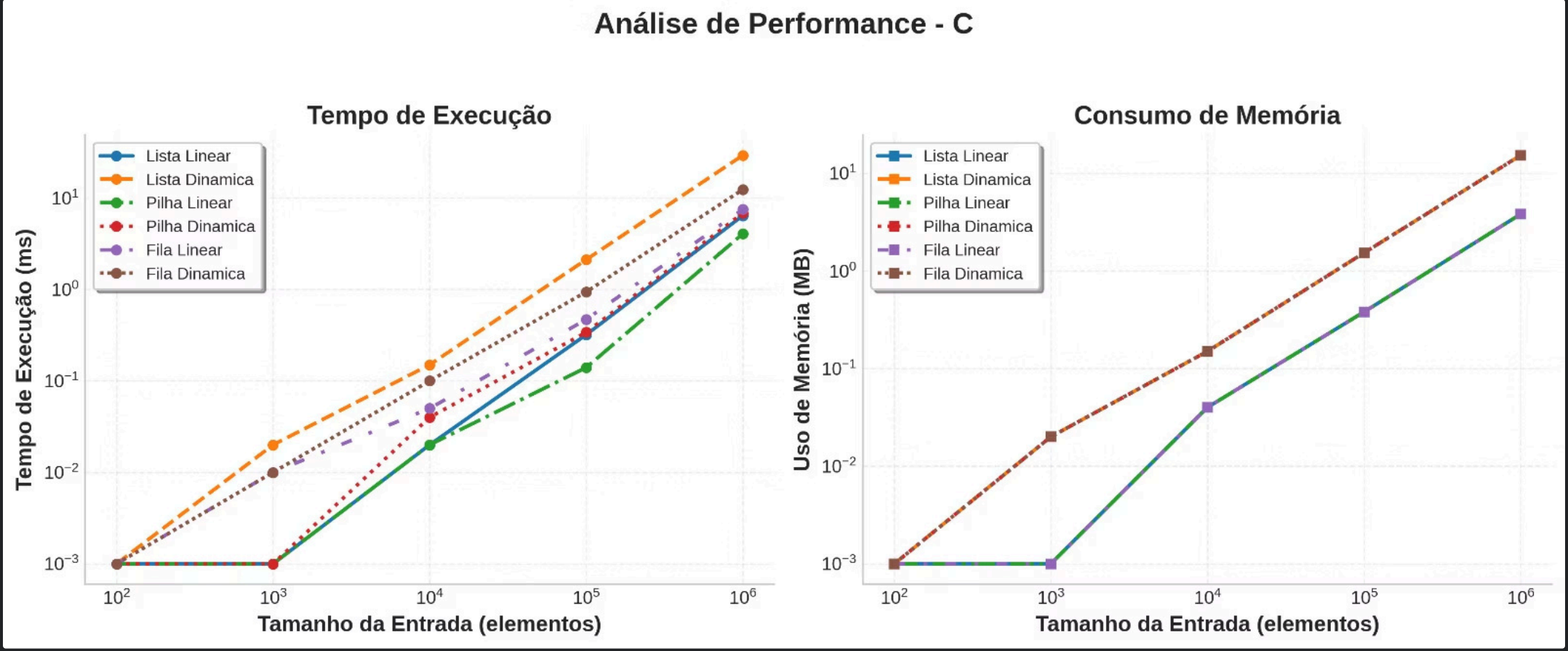
Comparação entre Linguagens - Estruturas Lineares x Dinâmicas



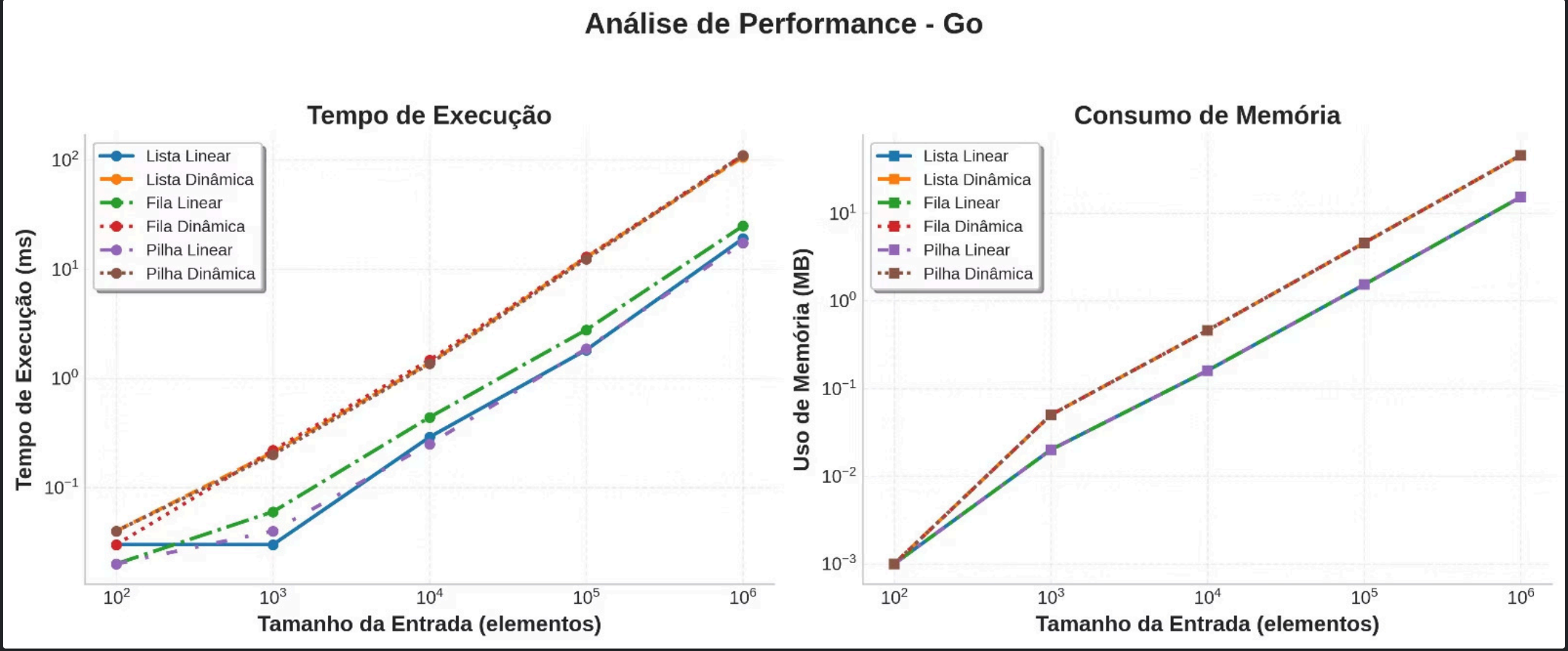
Análise de Performance - C++



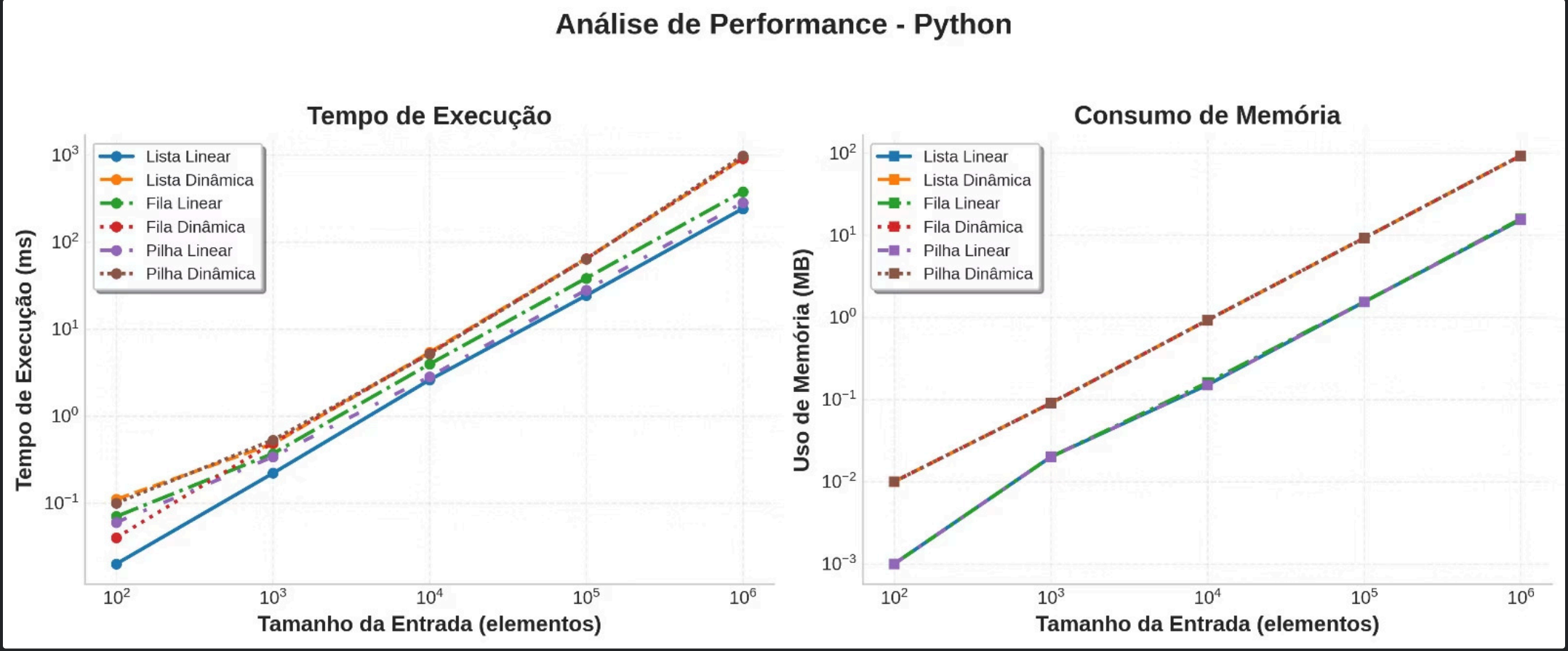
Análise de Performance - C



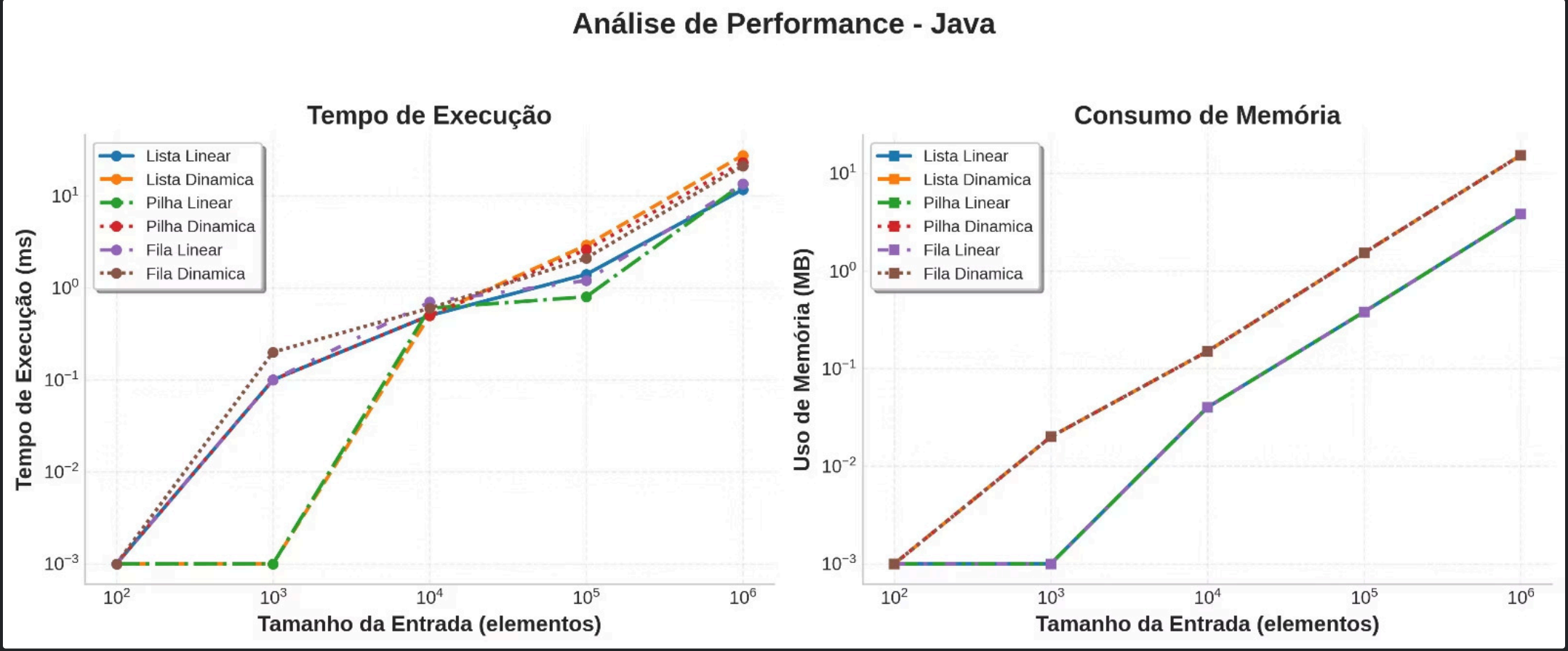
Análise de Performance - Go



Análise de Performance - Python



Análise de Performance - Java



Conclusão

Nesta apresentação, exploramos profundamente os algoritmos de ordenação não comparativos, destacando o Counting Sort e o Radix Sort. Discutimos suas origens e identificamos cenários de aplicação ideais, especialmente quando o intervalo de valores (K) se aproxima do número de elementos (N). Nossa análise de desempenho em diversas linguagens de programação revelou a superioridade de velocidade em C e C++, o equilíbrio oferecido por Go, e a tendência de menor desempenho em Python e Java, atribuída às suas camadas de abstração.

Concluímos que a escolha de um algoritmo e da linguagem mais apropriada transcende a mera complexidade teórica, exigindo a consideração atenta do contexto de execução, das características específicas dos dados e do objetivo final. Na prática, esses algoritmos demonstram ser ferramentas inestimáveis em sistemas que processam grandes volumes de dados inteiros ou categóricos, como na ordenação de registros, análise de logs, sistemas de contagem e em etapas de pré-processamento para algoritmos de aprendizado de máquina, onde a eficiência e a rapidez de resposta são fatores de importância crítica.

Obrigado!

Dúvidas e perguntas são bem-vindas!

Sinta-se à vontade para entrar em contato para mais informações ou para discutir o tema.