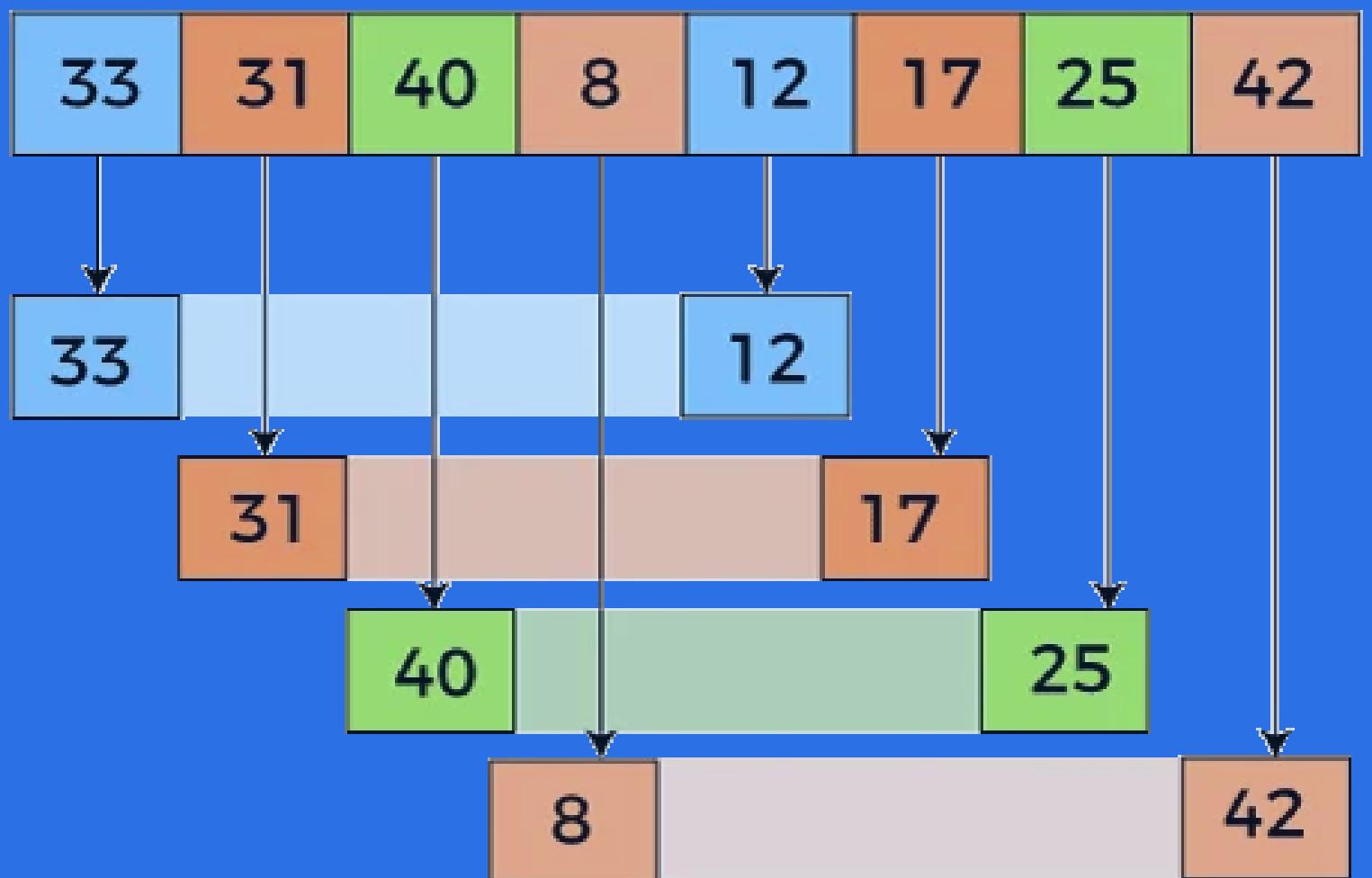


Ordenação Incrementais

Shell Sort e estratégias de Gaps



Sumário

- Insertion Sort, a base do Shell Sort
- Origem Shell Sort
- Passo a Passo do Shell Sort
- Entendendo o Gap
- Estratégias de Gap
- Shell Sort em Diferentes Linguagens
- Comparação com diferentes Estrutura de Dados
- Desafios e Lições
- Conclusão

Insertion Sort: Como Funciona



Antes de apresentar o Shell Sort, é fundamental recapitular o Insertion Sort, já que ele é a base conceitual do algoritmo.

Ideia Principal

Insere cada elemento na posição correta entre os já ordenados à esquerda.

Processo

Para $i = 1 \dots n-1$, “pega” $A[i]$ e desloca elementos maiores à direita até achar o lugar certo.

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1 \dots j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

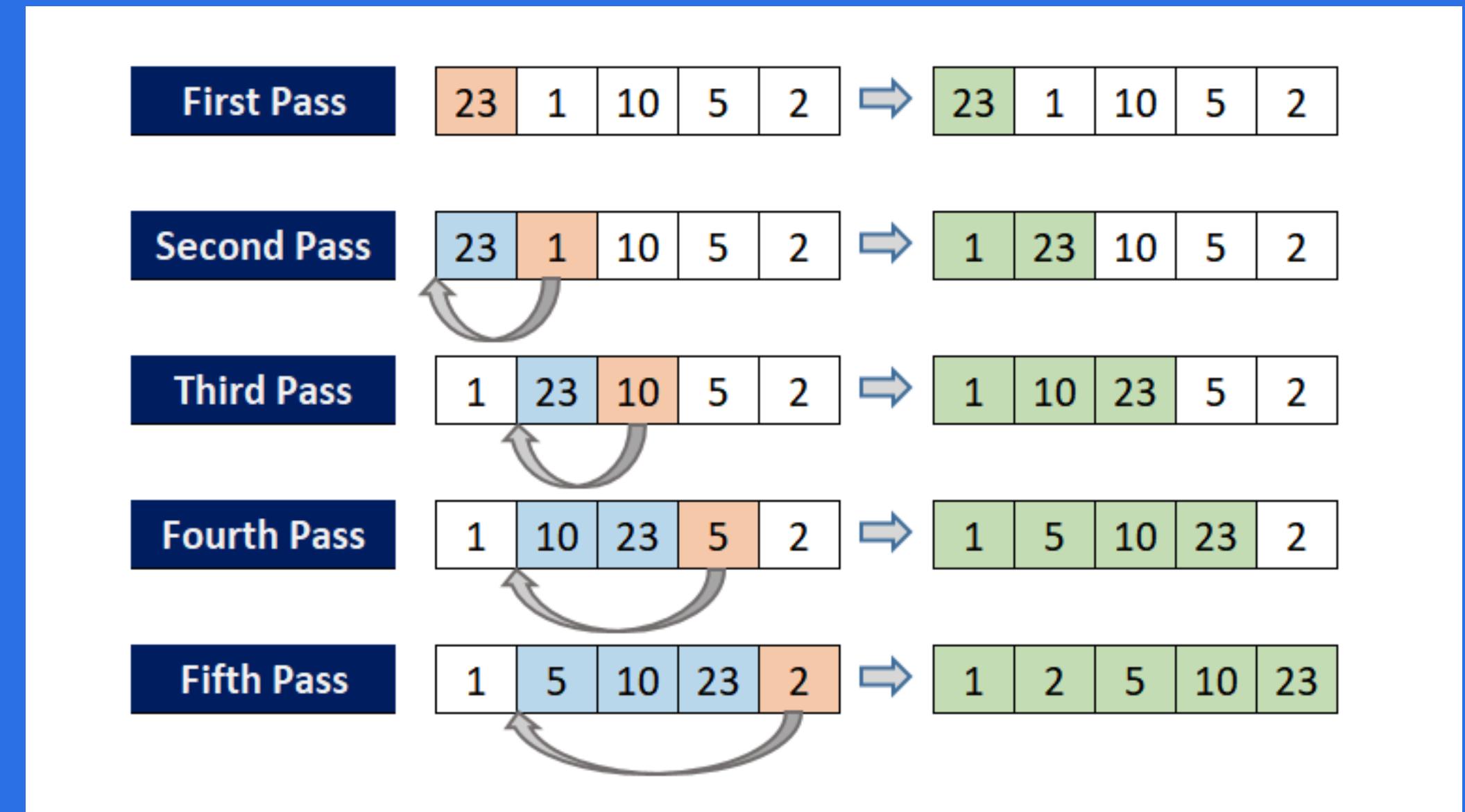
Insertion Sort: Desempenho & Exemplo

Complexidade

- Melhor caso (quase ordenado): $O(n)$
- Médio/pior caso (invertido): $O(n^2)$

Estabilidade & Memória

- Estável (mantém ordem de iguais)
- In-place (uso $O(1)$ de memória extra)



Gargalo do Insertion Sort

O Insertion Sort faz muitas trocas quando o vetor está desordenado, pois move cada elemento vizinho um a um, gerando custo quadrático em grandes listas.

Ideia de “saltos” (gaps)

Em vez de comparar apenas elementos adjacentes, o Shell Sort compara itens separados por um gap, reduzindo a distância que cada valor precisa percorrer.



Por que evoluir o Insertion Sort?

Pré-ordenação parcial

Cada passagem com gap alto organiza parcialmente o vetor, aproximando valores de suas posições finais antes da fase de gap=1.

Melhora no desempenho

Ao diminuir os deslocamentos mais longos antes do passo final, o número total de movimentos críticos cai, acelerando o tempo de ordenação.

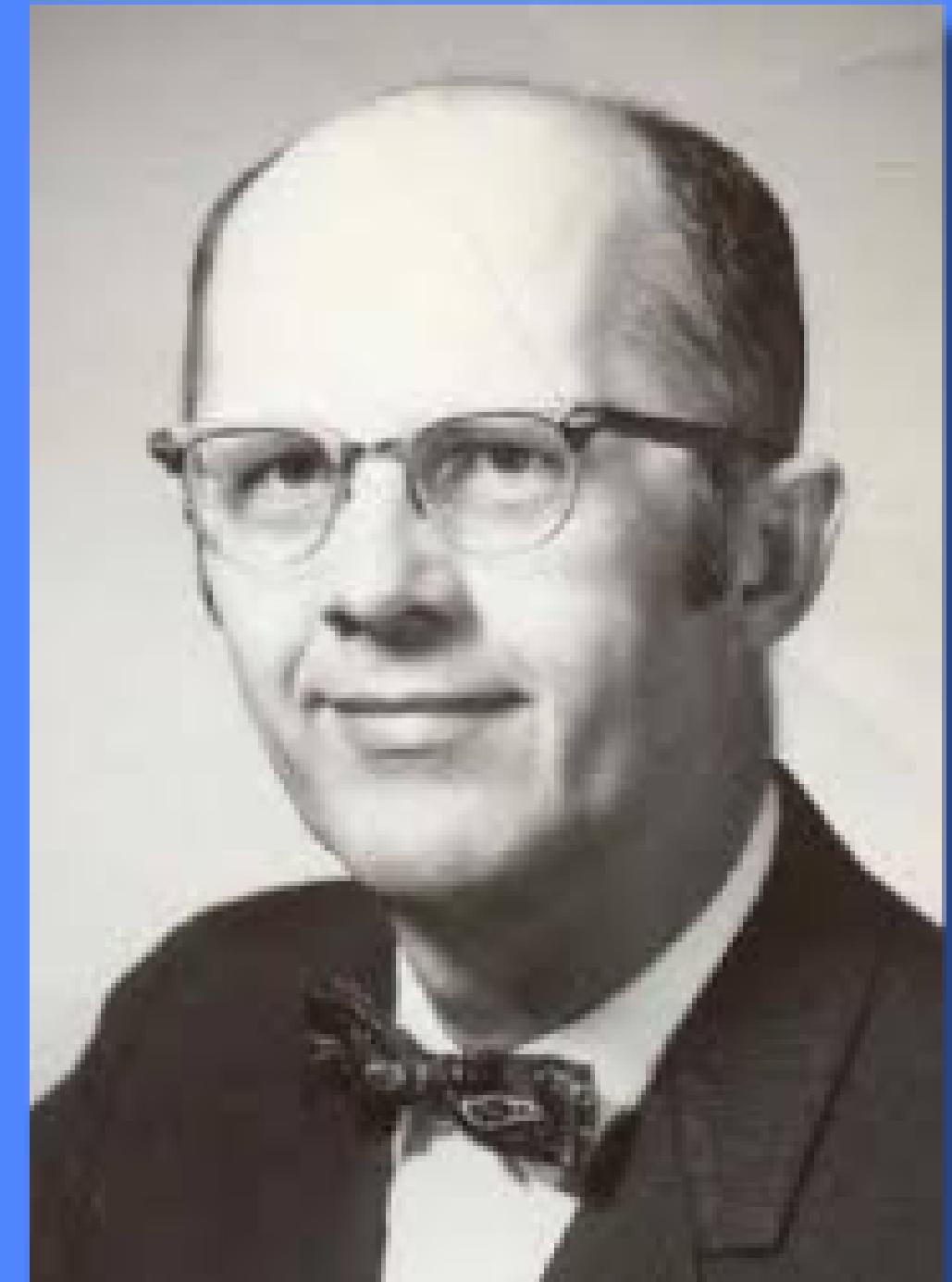


Origem do Shell Sort

O Shell Sort foi criado em 1959 por **Donald L. Shell**.

Foi o primeiro algoritmo a melhorar o Insertion Sort, quebrando a barreira de complexidade quadrática.

Ele usa uma sequência de gaps, que vão diminuindo até o algoritmo se comportar como um **Insertion Sort** final.





Visão Geral do Shell Sort

Ideia Principal

- Ordenar o vetor em “saltos” (gaps) para aproximar elementos distantes.
- Reduzir a desordem geral antes do Insertion Sort final.

Pseudocódigo



Shell Sort - Pseudocode

```
k = last/2          //compute original k value
loop (k not 0)
    current = k
    loop (current <= last)
        hold = list[current]
        walker = current - k
        loop (walker >= 0 AND hold < list[walker])
            list[walker+k] = list[walker]      //move larger element
            walker = walker - k             //recheck previous comparison
        end loop
        list[walker+k] = hold           //place the smaller element
        current = current + 1
    end loop
    k = k/2          //compute the new k value
end loop
```

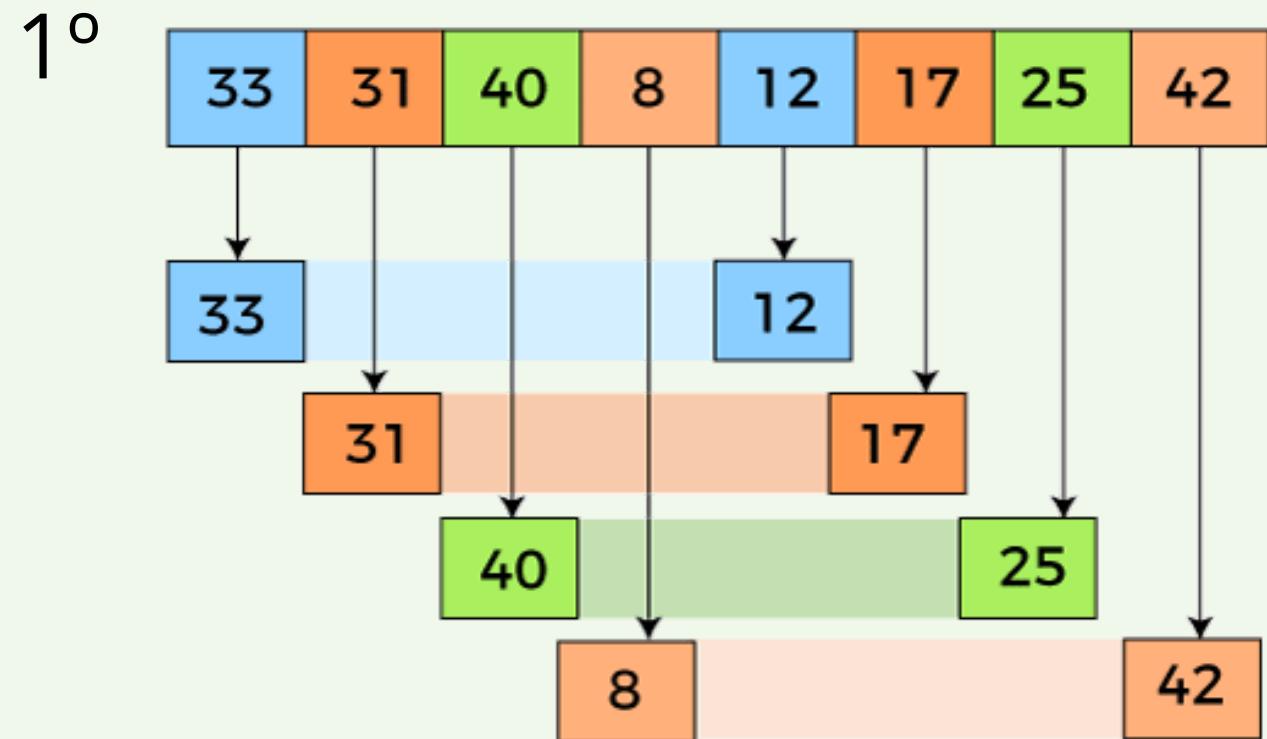
Passo a Passo

A seguir, apresentamos um passo a passo da ordenação utilizando o algoritmo Shell Sort com a estratégia de gap padrão ($n/2$). Os dados utilizados são os seguintes:

33	31	40	8	12	17	25	42
----	----	----	---	----	----	----	----

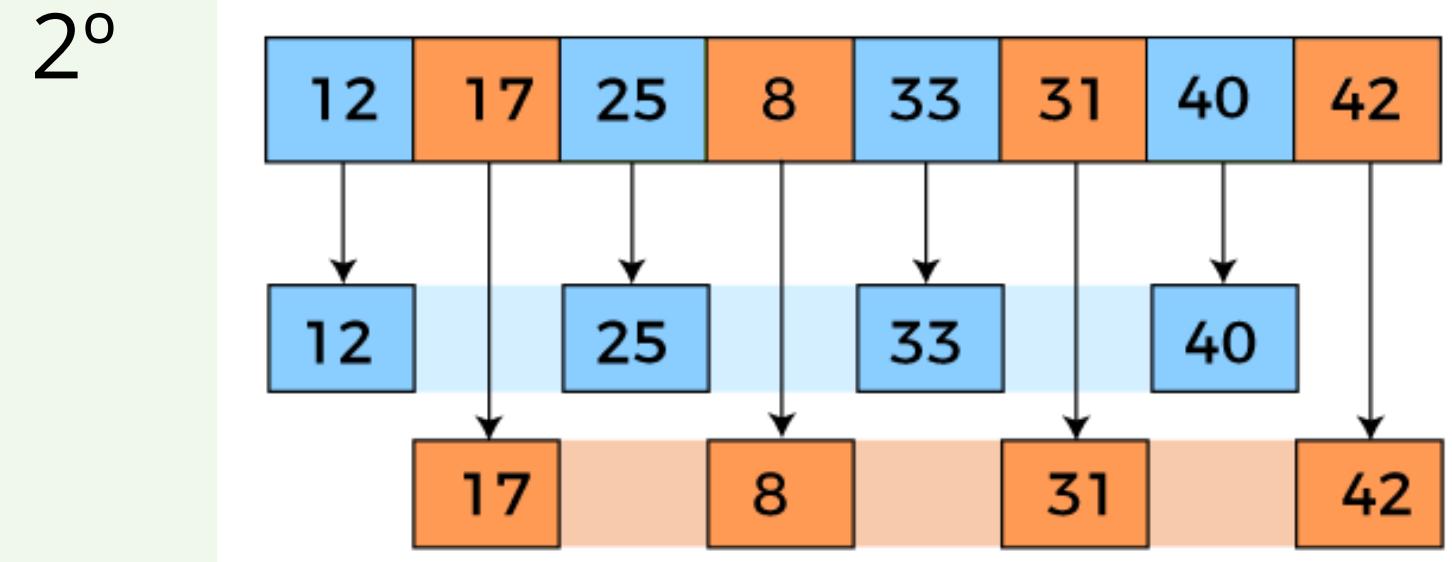
Passo a Passo

Gap $n/2 \Rightarrow 8/2 = 4$



12	17	25	8	33	31	40	42
----	----	----	---	----	----	----	----

Gap 2



12	8	25	17	33	31	40	42
----	---	----	----	----	----	----	----

Insertion Sort

3º

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Resumo do Passo a Passo

- ◆ **Define-se o gap**

Começa com $\text{gap} = n/2$

Gap é a distância entre os elementos comparados

- ◆ **Ordena usando os gaps**

Compara elementos separados por gap

Aplica uma versão do Insertion Sort

Cada sublista é parcialmente ordenada

- ◆ **Reduz o gap**

Gap é reduzido a cada rodada ($\text{gap} = \text{gap}/2$)

Repete o processo com os novos gaps

- ◆ **Gap final igual a 1**

Última etapa é um Insertion Sort

Agora os dados já estão quase ordenados

Fica muito mais eficiente

Entendendo o “Gap” no Shell Sort

O que é “Gap”?

No Shell Sort, o gap é o intervalo entre os elementos que serão comparados e ordenados. Ao invés de comparar elementos lado a lado, como no Insertion Sort, o algoritmo começa comparando elementos mais distantes entre si. Isso ajuda a mover valores grandes mais rapidamente para perto da posição correta.

Como ele muda:

O algoritmo começa com um gap grande e vai diminuindo aos poucos, geralmente dividindo por 2 ou **segundo uma sequência específica**. Quando o gap chega a 1, o Shell Sort realiza um Insertion Sort final, mas com os dados já quase ordenados, o que melhora bastante a eficiência.

Entendendo o “Gap” no Shell Sort

Por que reduzir o gap?

Reducir o gap gradualmente permite que os elementos se aproximem das suas posições corretas aos poucos. Com gaps maiores no início, valores distantes são organizados rapidamente. À medida que o gap diminui, o algoritmo refina a ordenação até que, com gap 1, o vetor esteja quase ordenado e o Insertion Sort final seja muito mais eficiente.

Impacto no desempenho:

A escolha e redução do gap afetam diretamente o número de comparações e trocas feitas pelo algoritmo. Estratégias bem planejadas de gaps podem transformar o Shell Sort de um algoritmo lento ($O(n^2)$) em um algoritmo competitivo com desempenho próximo ao dos melhores métodos de ordenação, como Quick Sort ou Merge Sort.

Estratégias Clássicas de Gaps

Shell (Original):

Divide o tamanho do vetor por 2 a cada passo. Simples, mas pouco eficiente ($O(n^2)$).

Hibbard

Na sequência de Hibbard, os gaps são definidos pela fórmula $gap = 2^k - 1$, onde k é um número inteiro positivo (ex: 1, 2, 3, ...). Isso gera gaps como 1, 3, 7, 15, etc.

Estratégias Avançadas de Gaps

Sedgewick:

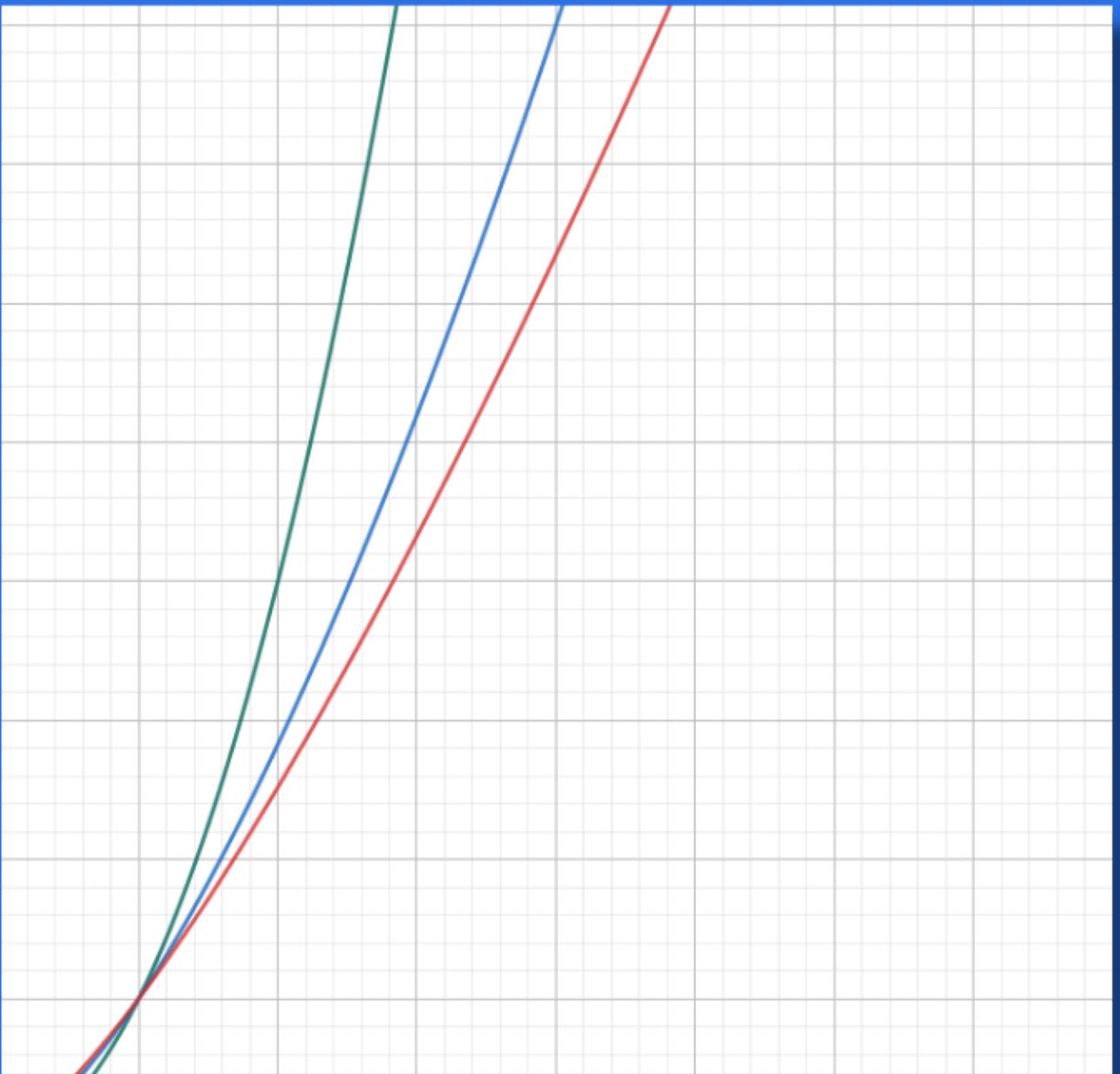
Combina fórmulas de potências de 2 e 3. Tem desempenho prático muito bom e complexidade $O(n^{4/3})$.

Pratt:

Usa todos os números da forma $2^p * 3^q$. Excelente desempenho, mas difícil de implementar.

Comparação Teórica das Estratégias de Gaps

Estratégia	Complexidade no pior caso
Shell (original)	$O(n^2)$
Knuth	$O(n^{3/2})$
Sedgewick	$O(n^{4/3})$



Sequência de Knuth



Escolhemos a sequência de Knuth para os testes por ser, ao mesmo tempo, eficiente e simples de implementar.

Fórmula da Sequência

A sequência é definida por $(3^k - 1) / 2$, gerando valores como 1, 4, 13, 40, 121... Esses gaps crescem de forma equilibrada e controlada, favorecendo uma boa ordenação em cada fase.

Desempenho prático excelente:

Em testes reais, a sequência de Knuth apresenta tempos de execução muito bons, superando a sequência original de Shell e chegando perto de estratégias mais avançadas.

Boa distribuição de gaps

Os gaps aumentam gradualmente, começando com valores pequenos e atingindo números maiores, permitindo uma ordenação progressiva e eficiente.

Simples de implementar:

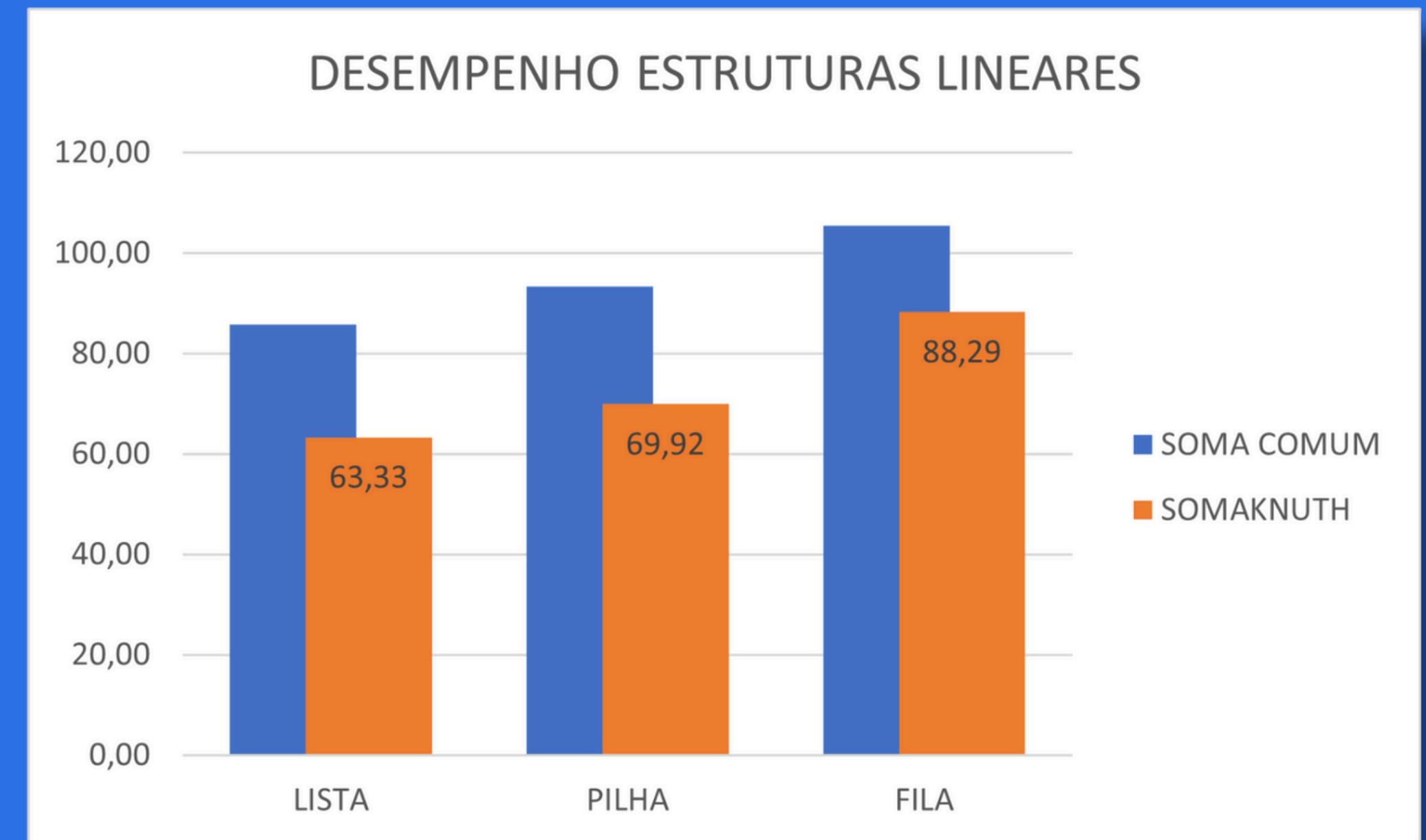
Pode ser gerada com um laço simples em qualquer linguagem. Não exige fórmulas complexas nem cálculos pesados.

Comparação Prática

📍 Observações:

O Shell Sort com Knuth é consistentemente mais rápido em todas as estruturas.

O ganho de desempenho é mais visível em **listas**, chegando a mais de 26% de economia no tempo de execução.

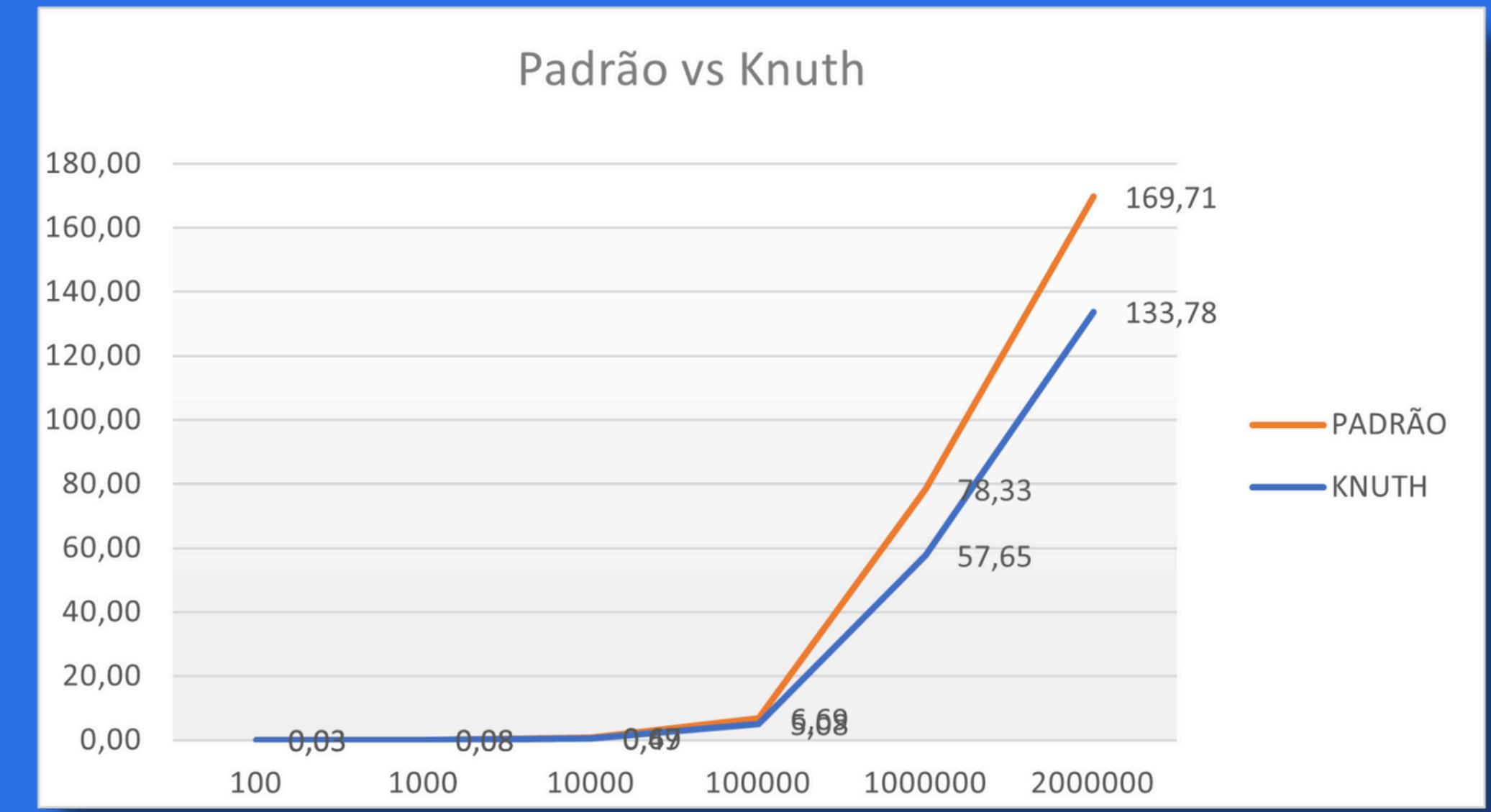


Soma total dos tempos obtida a partir da média de 100 execuções por caso de teste, utilizando a linguagem Java. Comparação entre Shell Sort tradicional e com sequência de Knuth.

Comparação Prática

📌 A sequência de Knuth apresentou melhor desempenho em diferentes volumes de dados, especialmente a partir de 10 mil elementos. Comparada à sequência padrão ($n/2$), mostrou-se mais eficiente e com comportamento mais linear em entradas maiores.

Quantidade de Dados	Variação
100	7%
1000	0%
10000	27%
100000	24%
1000000	26%
2000000	21%



Shell Sort em Diferentes Linguagens de Programação

Um dos nossos objetivos é analisar o desempenho do Shell Sort em diferentes linguagens de programação. A partir deste ponto, passaremos a mostrar como o algoritmo se comporta em cada linguagem, considerando fatores como tempo de execução e uso de memória.

Comparar o Shell Sort em diferentes linguagens nos permite entender como o desempenho do algoritmo varia de acordo com o paradigma, o tempo de execução, o consumo de memória e o nível de abstração de cada linguagem. Essa análise é importante para escolher a melhor ferramenta conforme o contexto e os requisitos do projeto.

Linguagens Escolhidas

C/C++

Julia

Java

Python

Comparação Prática

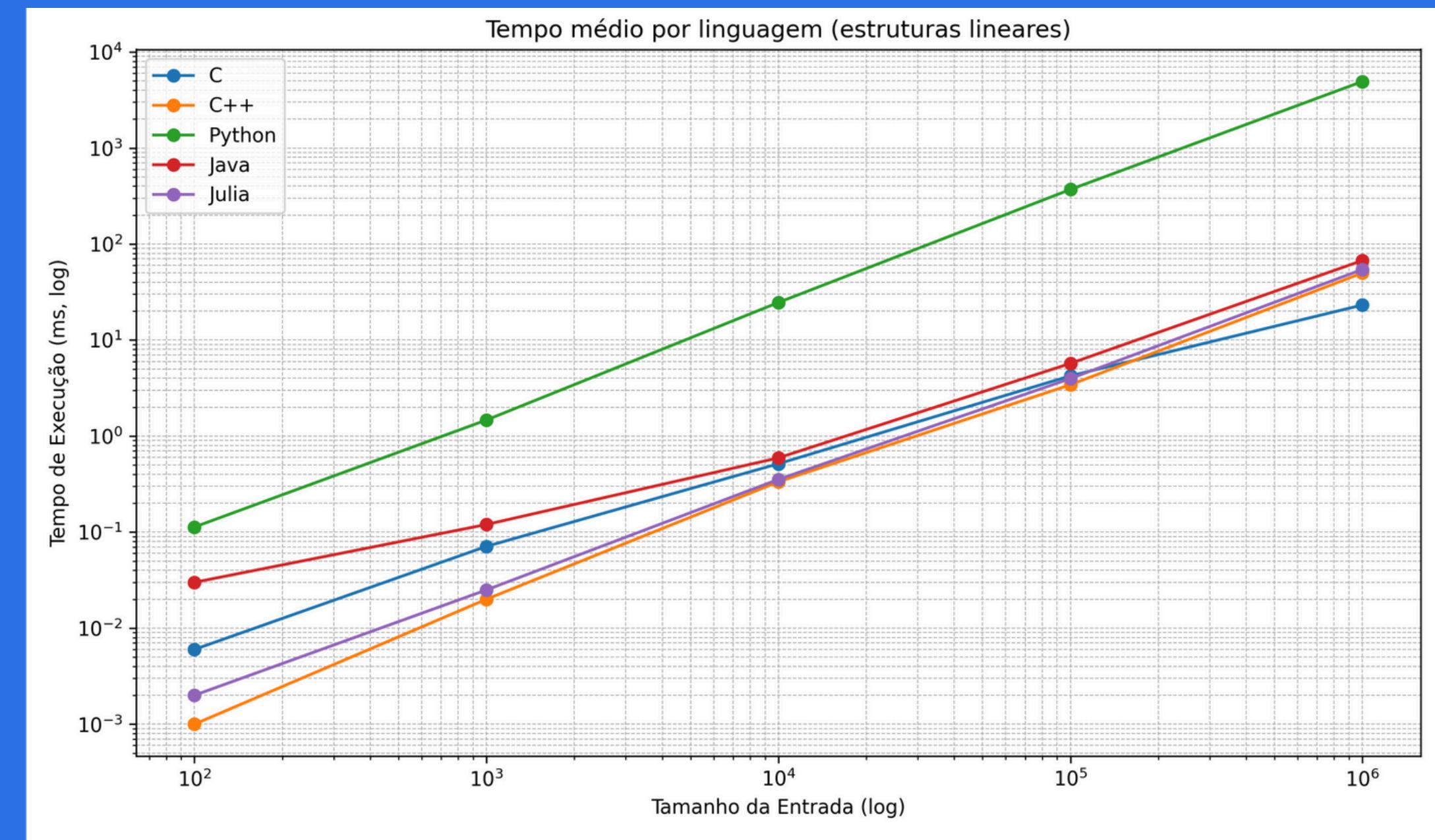
Todas as linguagens apresentaram desempenho melhor do que com estruturas dinâmicas, com crescimento mais suave e tempos menores.

C e C++ foram novamente os mais rápidos, aproveitando o acesso direto e contíguo à memória dos arrays.

Julia manteve desempenho próximo de C e C++, mostrando boa eficiência com vetores nativos.

Java teve desempenho consistente, mas ainda atrás dos compilados, devido ao overhead da JVM.

Python, mesmo com otimizações internas, continuou com o pior desempenho, especialmente após 10.000 entradas — reflexo do custo de abstração e tipagem dinâmica.

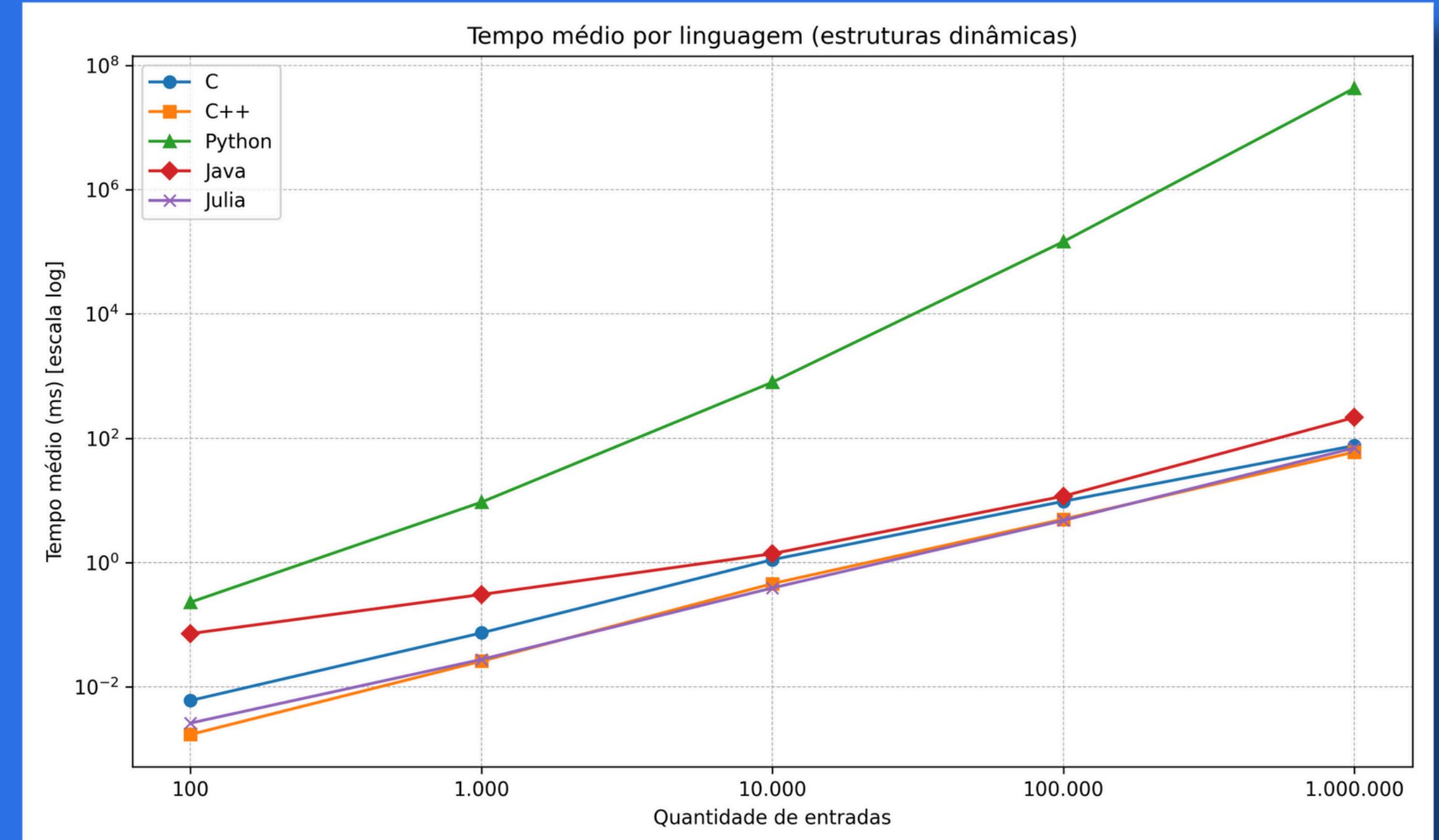


Comparação Prática

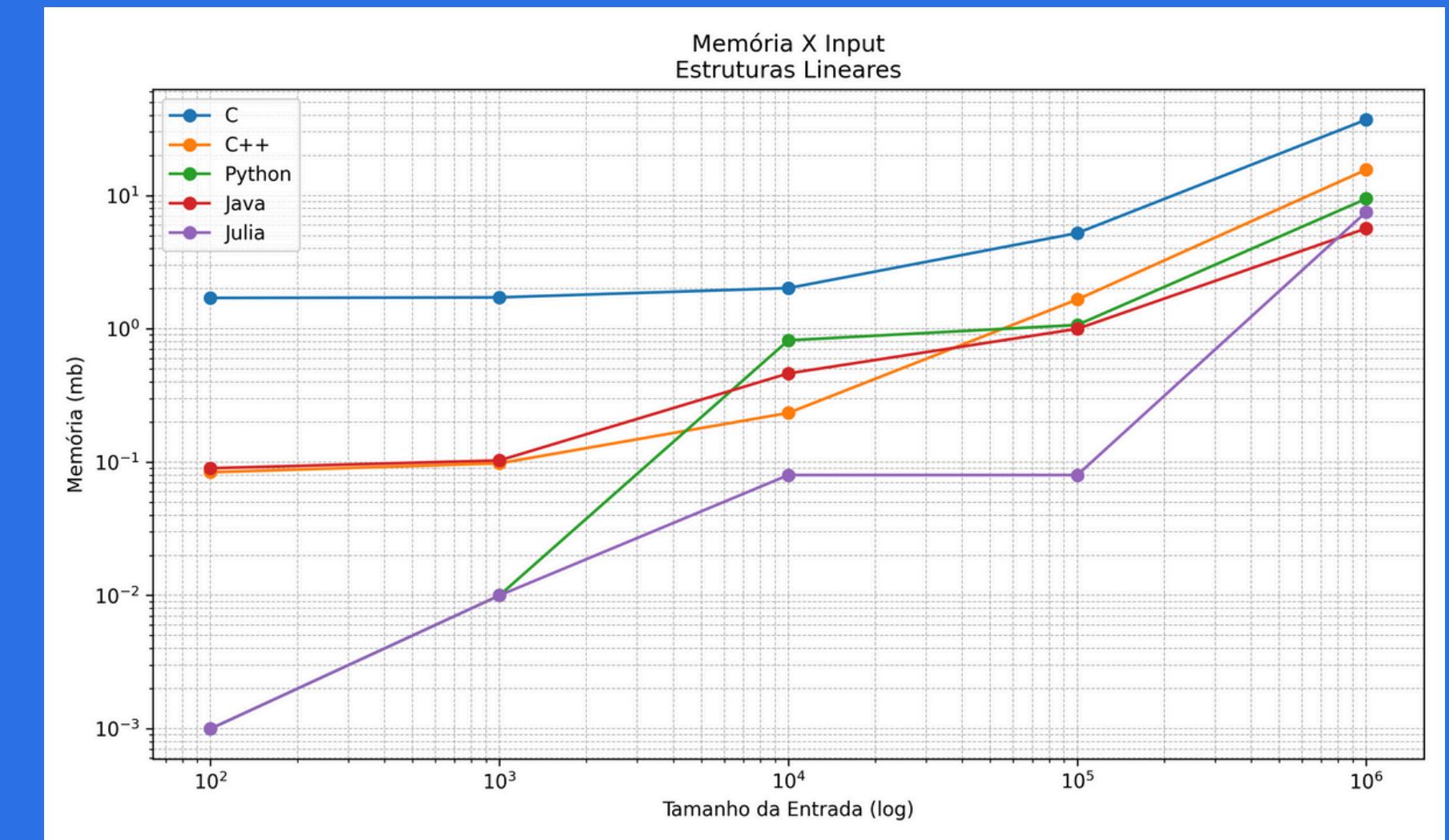
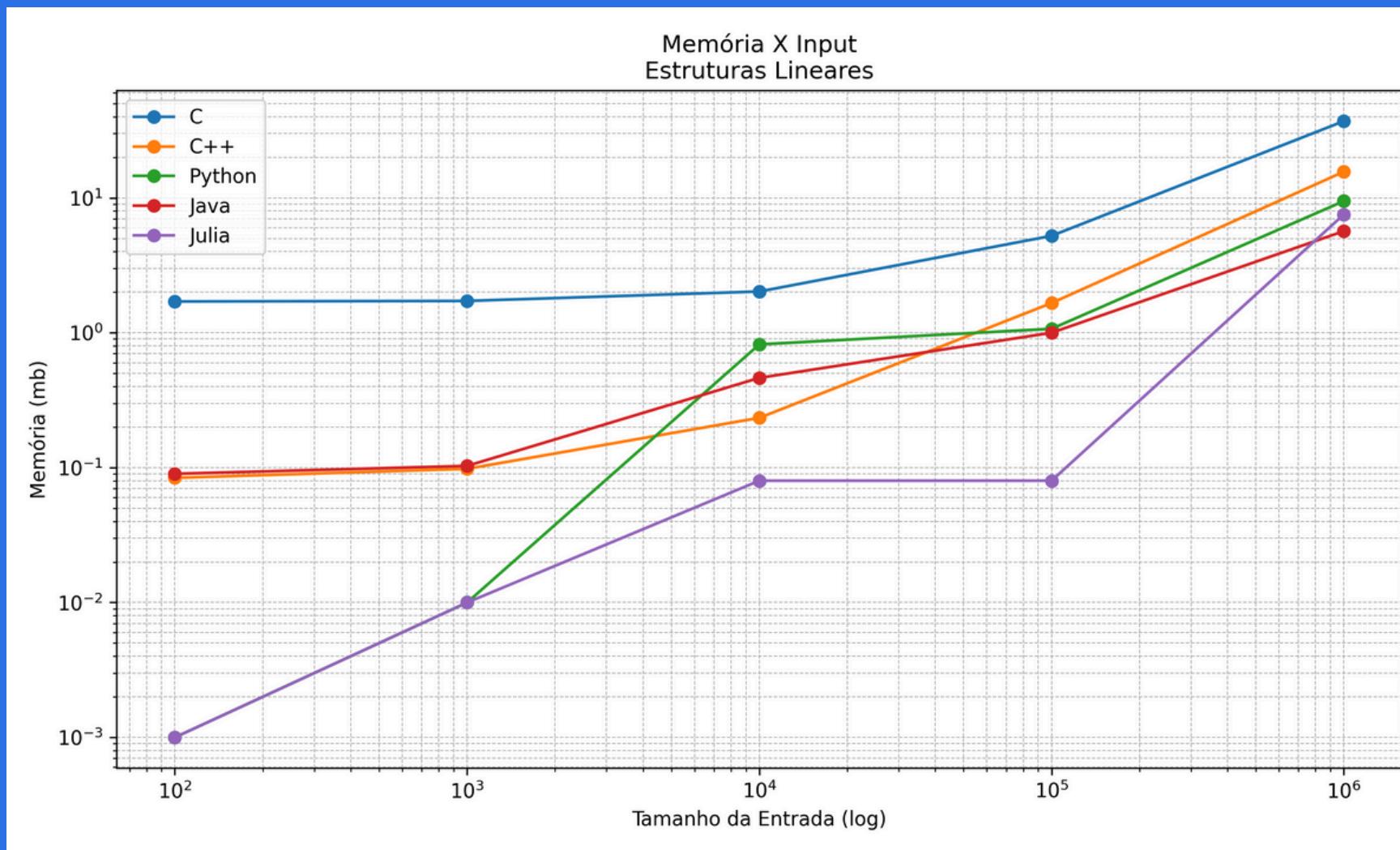
Python apresentou o pior desempenho geral, com tempo de execução crescendo rapidamente conforme o volume de dados aumenta.

Java ficou logo acima, mas ainda foi mais lento que C, C++ e Julia em grandes volumes, provavelmente pelo custo da **reconstrução** de objetos dinâmicos.

C, C++ e Julia tiveram tempos bem próximos e os melhores resultados, pois são linguagens compiladas, com acesso direto à **memória** e menos sobrecarga no tratamento de ponteiros.



Uso de Memória



Comparação Geral

Python foi a mais lenta

- Demorou mais para rodar.
- **Overhead** do interpretador e tipagem dinâmica tornam as operações mais lentas

Java teve desempenho médio

- Rodou mais rápido que o Python, mas ficou atrás de C/C++ e Julia.
- Usa mais memória por precisar organizar objetos em segundo plano
- Com **1 milhão** de linhas, foi a que **menor consumiu** memória, e também foi a que teve o uso mais linear de memória

C, C++ e Julia foram as melhores em Tempo de Execução

- Trabalham direto com o bloco de memória, sem etapas extras.



Estruturas Lineares e Dinâmicas Utilizadas

Os testes mais avançados foram realizados em Java, por ser a linguagem com melhor suporte às diferentes estruturas utilizadas. Cada cenário foi executado 100 vezes em média para garantir resultados mais precisos e confiáveis.

As duas versões receberam a mesma lógica de Shell Sort, permitindo comparar o comportamento.

Estruturas Dinâmicas (baseadas em ponteiros):

List<Double> (Lista)

Stack<Double> (Pilha)

Queue<Double> (Fila)

São mais flexíveis, mas com acesso mais lento

Estruturas Lineares (baseadas em arrays):

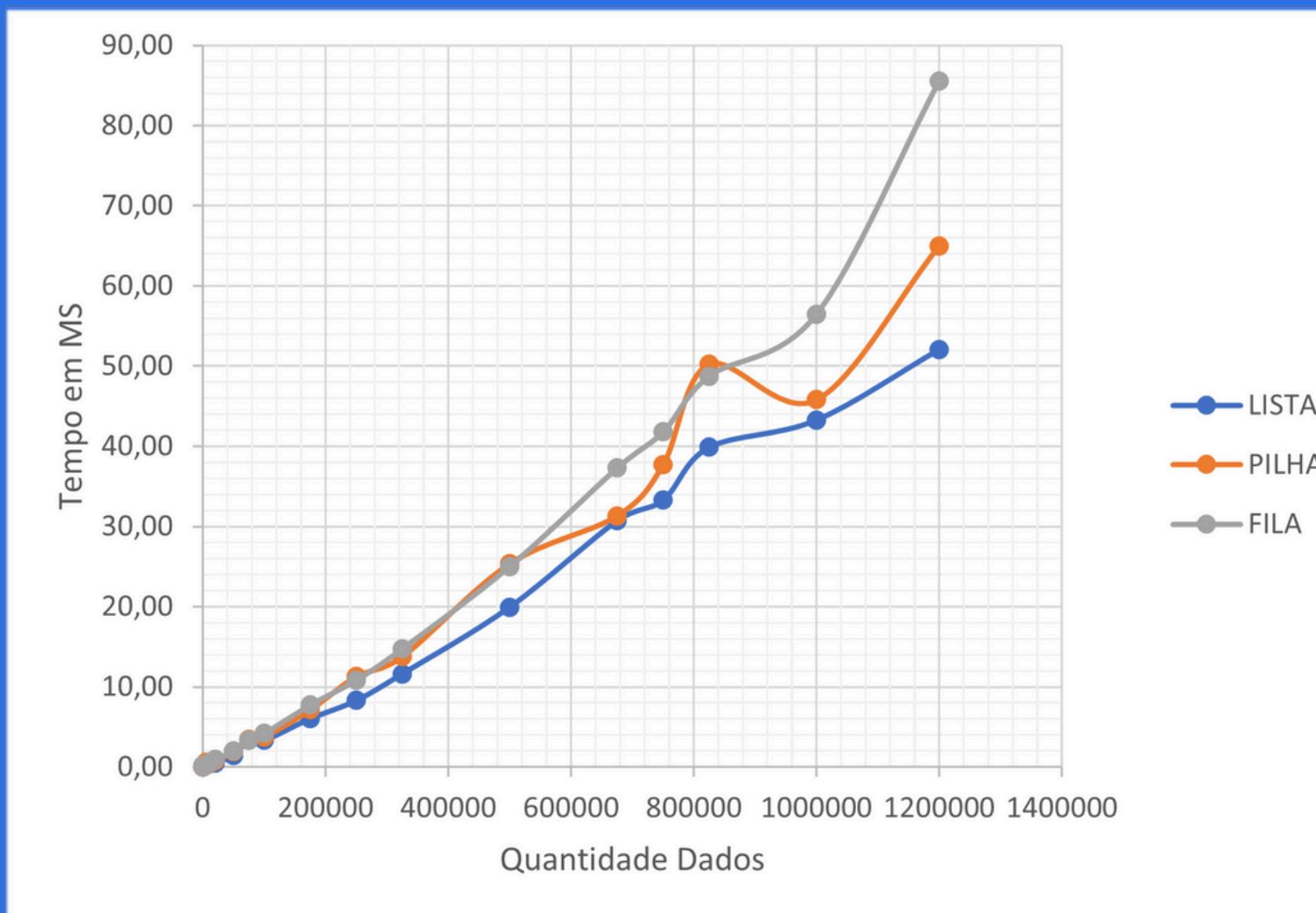
Vetores (double[])

Pilha e fila implementadas com arrays
(PilhaLinear, FilaLinear)

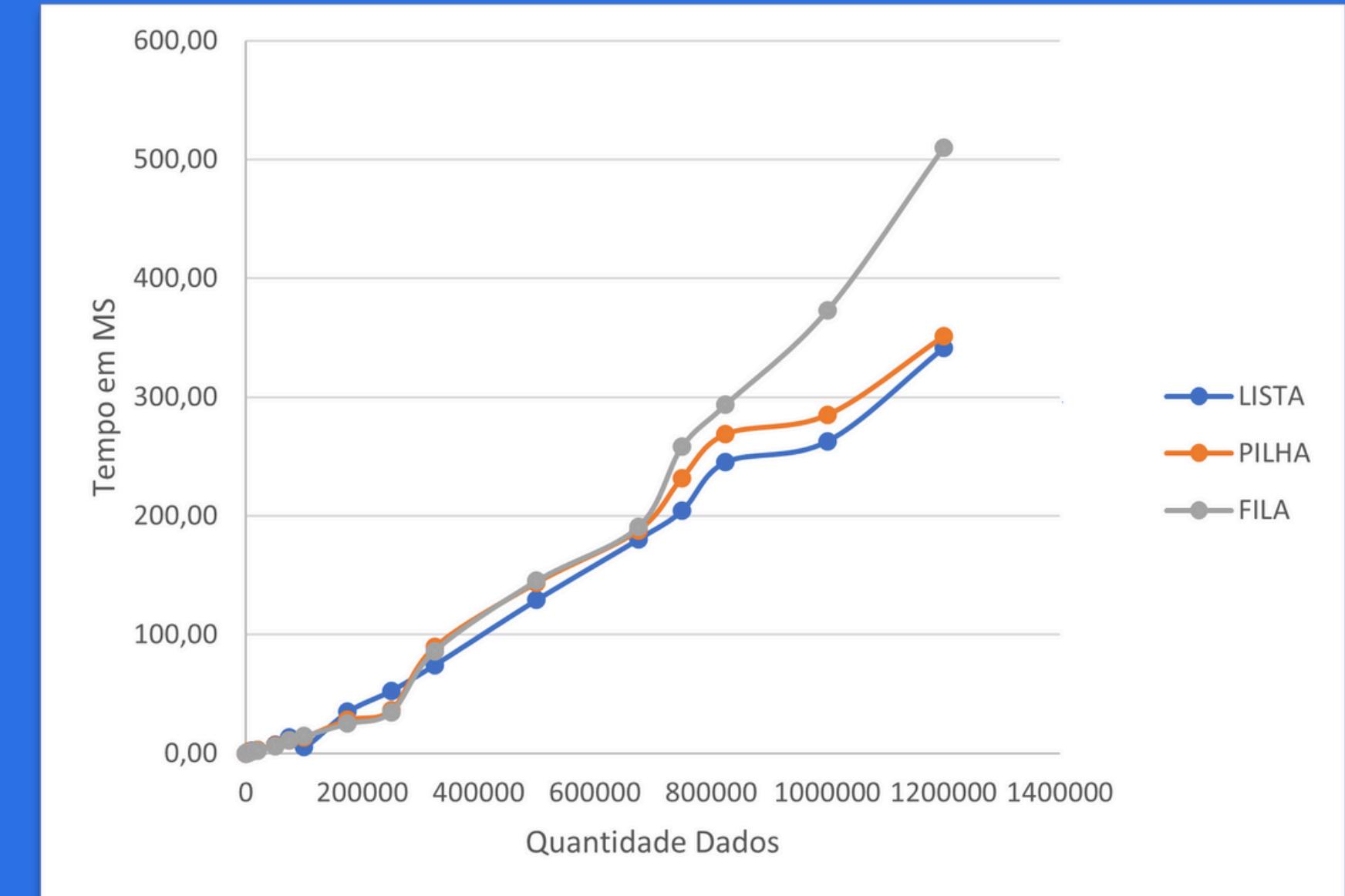
Permitem acesso direto aos elementos

Comparação Prática entre Estruturas

Estrutura Linear



Estrutura Dinâmica



Análise de Desempenho - Estrutura Linear vs Dinâmica

Estrutura Linear

Menor tempo de execução em todos os testes.

Crescimento gradual e estável com o aumento da quantidade de dados.

A lista (vetor) foi a mais eficiente, seguida de pilha e depois fila.

Estrutura Dinâmica

Apresentaram tempos significativamente maiores.

A fila dinâmica foi a mais lenta, devido ao custo de acesso e realocação via ponteiros.

A pilha dinâmica teve desempenho melhor que a fila, mas ainda inferior à lista dinâmica.

A lista dinâmica também teve queda de desempenho em relação à linear, mas menos acentuada.



Conclusões

- Estruturas lineares são mais eficientes para o algoritmo Shell Sort.
- A sobrecarga de gerenciamento de ponteiros nas estruturas dinâmicas impacta negativamente o desempenho.
- Para ordenação de grandes volumes de dados em Java, vetores (listas lineares) são a melhor escolha.

⚠️ Desafios e Limitações do Estudo

Seleção de Sequência de Gaps

Só testamos a sequência tradicional e Knuth
Outras (Sedgewick, Tokuda, Pratt)
podem ter performance ainda melhor
em certos tamanhos de dado

Condições de Teste

Como os dados foram coletados em diferentes máquinas, pode haver variações nos resultados devido a diferenças de hardware, como processador, quantidade de RAM, sistema operacional, ou até mesmo carga de uso no momento dos testes.

Lições Aprendidas

Valorização da Sequência de Knuth

Ganho consistente de 20 %-30 % em cenários reais

Reforça a importância de escolher bons gaps, não só “ $n/2$, $n/4\dots$ ”

Adequação de Estrutura de Dados

Vetores são ideais para ordenação in-place; listas encadeadas só se realmente necessárias

Em sistemas com inserções/remoções frequentes, pode valer a pena o trade-off

Futuras Extensões

- Testar outras sequências experimentais (Sedgewick, Pratt...)
- Paralelizar as passagens (multithreading) para aproveitar múltiplos núcleos
- Comparar Shell Sort com algoritmos híbridos nativos (TimSort, introsort)



Aplicações e Recomendações

Use Shell Sort com Knuth quando:

Você precisa de ordenação rápida e simples.

O dataset cabe em memória e pode ser armazenado em vetor.

O custo de implementação precisa ser baixo.

Evite usar Shell Sort:

Em estruturas baseadas em ponteiros sem conversão para array.

Quando a estabilidade do algoritmo for essencial (Shell Sort não é estável).

Quando algoritmos híbridos como TimSort estiverem disponíveis.

Conclusão Geral

Shell Sort é eficiente, especialmente com estrutura linear (vetor).

A sequência de Knuth melhora o desempenho em 20% a 30%, principalmente em grandes volumes de dados.

Listas, filas e pilhas dinâmicas têm maior custo de tempo devido à manipulação de ponteiros.

O estudo demonstrou que o método de Shell é uma alternativa eficiente para ordenação incremental, superando as limitações do Insertion Sort por meio de sequências de gaps estratégicas;

O Shell Sort é uma solução equilibrada entre desempenho e simplicidade, com resultados significativamente melhores em linguagens de baixo nível e com a escolha adequada de gaps.

Obrigado
duvidas?