

Métodos de Ordenação

Tim Sort

Pedro Galvão
Lucas Roseno
João Faria
Igor Peixoto

Julho 2025
9:00AM - 6:00PM

História do TimSort

Desenvolvido por Tim Peters em 2002, o **TimSort** é
► um método híbrido de ordenação de dados.
Derivado do **MergeSort** e **InsertionSort**

Implementada inicialmente em **Python**, mas hoje
► em dia linguagens mais modernas como Rust e
Swift, adotam o TimSort como algoritmo de
ordenação padrão

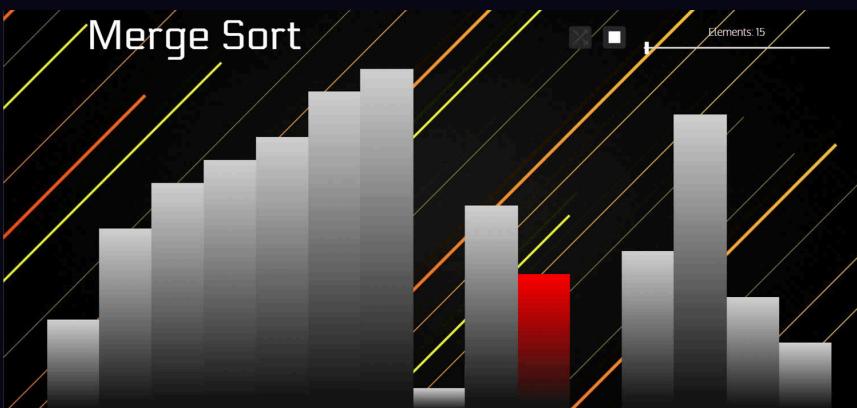
Utilizado também como mecanismo de
ordenação padrão no **Android** e no **V8** para tipos
► não primitivos de dados, como objetos, garantindo
estabilidade e agilidade



Sobre a Fusão ⚡

Merge Sort

- 01 Divide e conquista
- 02 Requer espaço auxiliar
- 03 Ideal para grande volume de dados
- 04 Lento para pequenos vetores



Complexidade garantida $O(n \log n)$

Insertion Sort

- 01 Se aproveita
- 02 In-place
- 03 Muito eficiente em vetores pequenos
- 04 Lento para grandes vetores

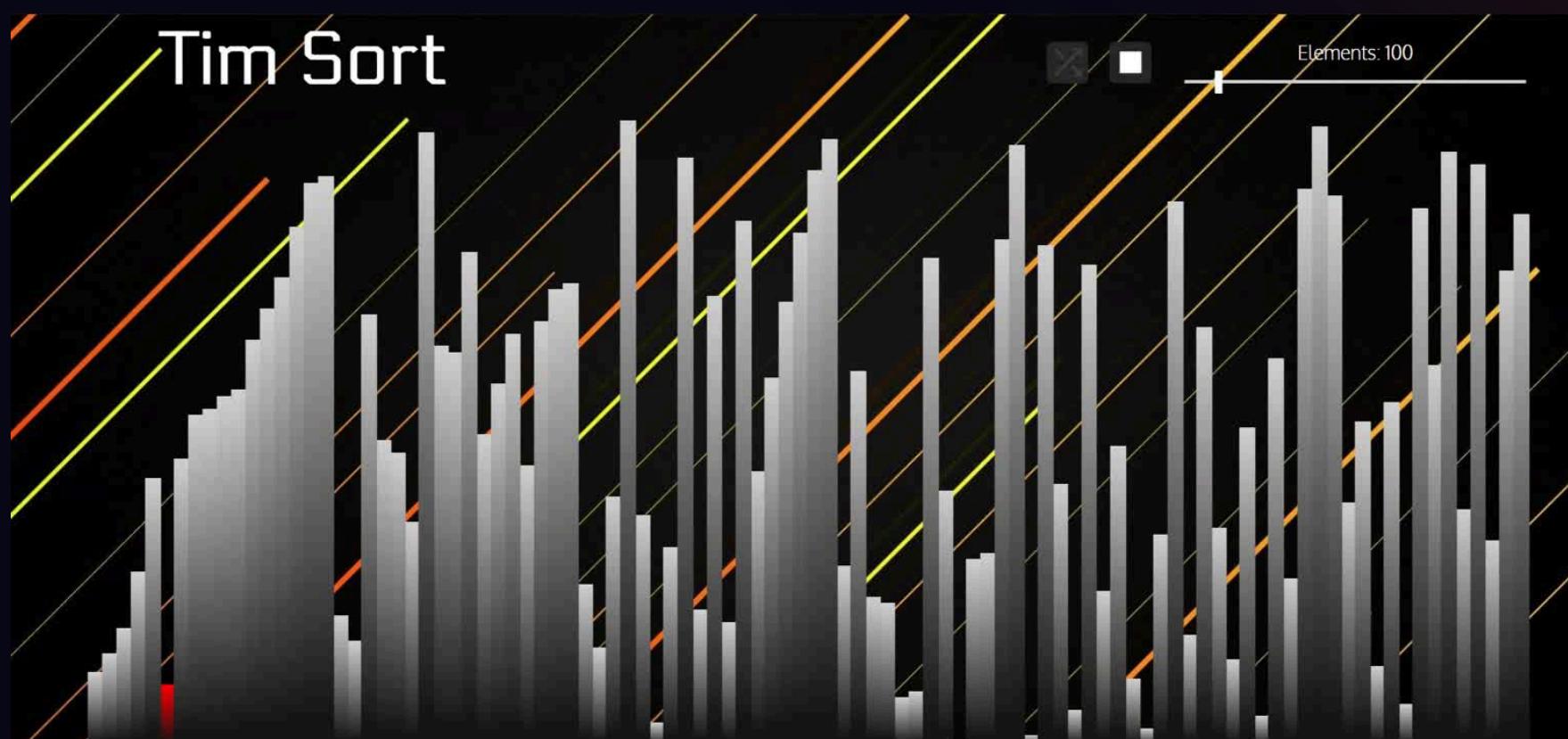


Melhor caso $O(n)$

Assim nasce o TimSort



“Pequenas partes ordenadas com rapidez. Grandes partes fundidas com eficiência.”



- ▶ Detecta runs ordenadas
- ▶ Ajusta o tamanho da run até $\geq \text{MIN_RUN}$
- ▶ Usa InsertionSort nas runs
- ▶ Faz Merge Inteligente das runs
- ▶ Aproveita ordenação parcial do array
- ▶ Extremamente eficiente em dados reais

Entendendo o Algorítimo



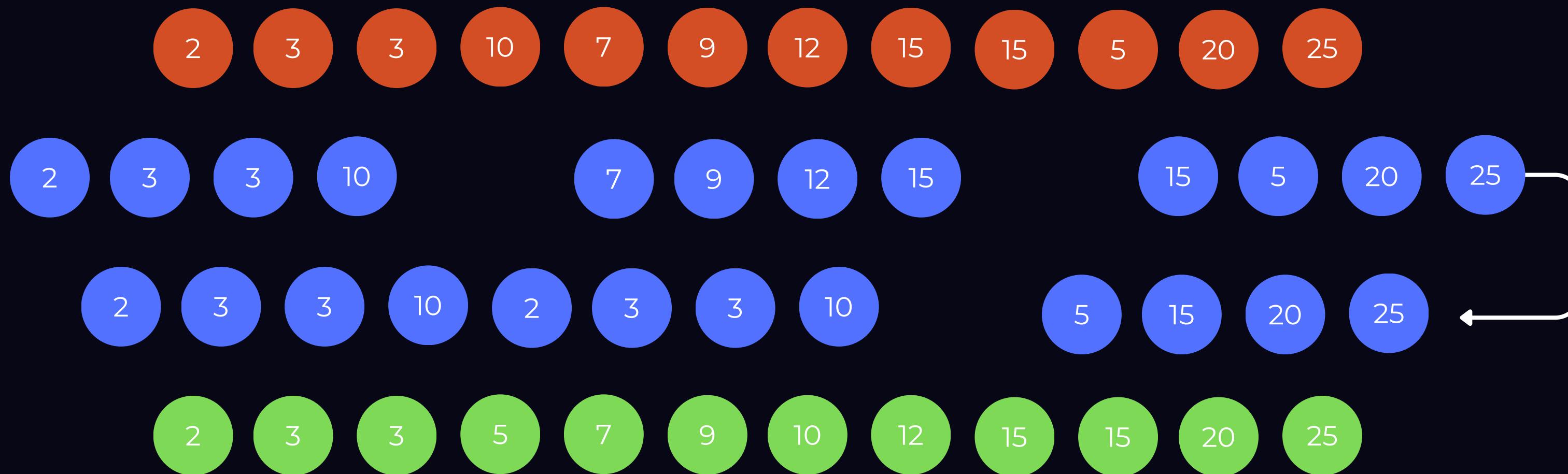
Fase I : Identificando Runs

Objetivo: Criar runs de tamanho mínimo para otimizar as próximas intercalações.

- ▶ Define MIN_RUN (Geralmente 32)
- ▶ Se a run for menor que MIN_RUN, estende.
- ▶ Percorre o array identificando runs naturais
- ▶ Ordena runs desordenadas
- ▶ Se a run for decrescente, inverte.



Entendendo o Algorítimo



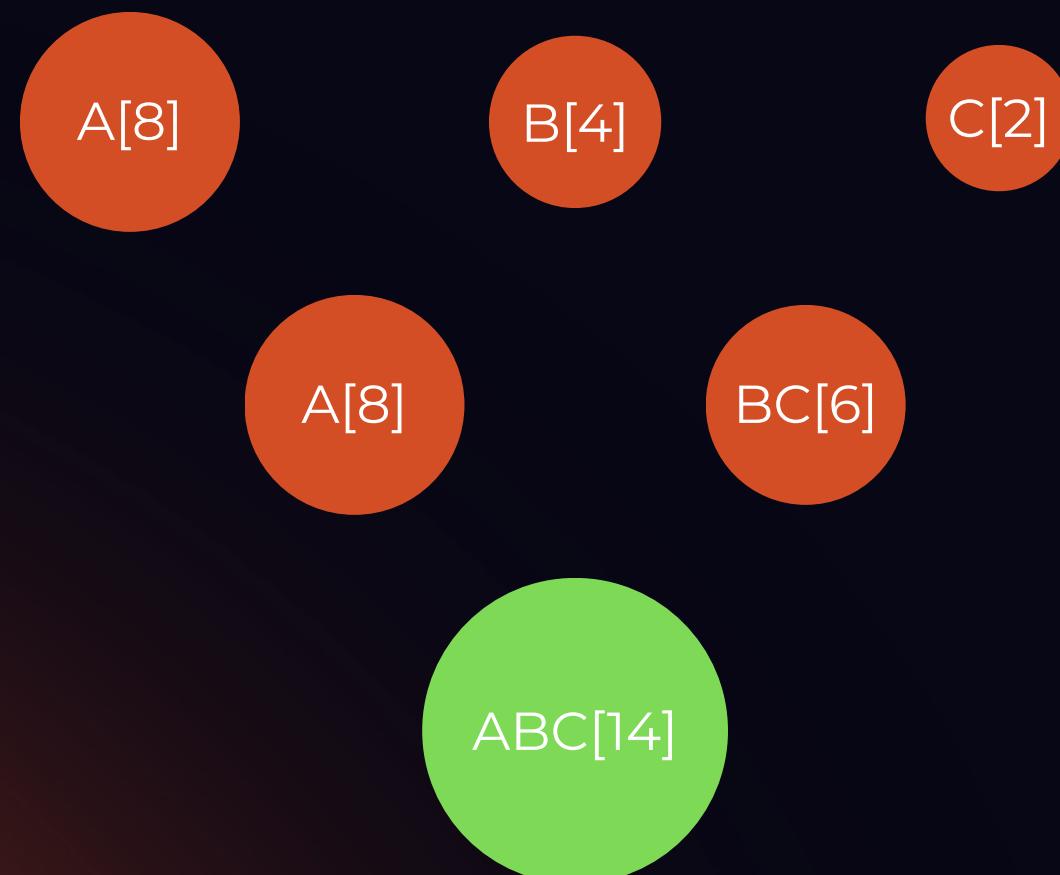
Entendendo o Algorítimo



Fase II : Intercalando Runs Intelligentemente

Objetivo: Mesclar as runs ordenadas que temos de maneira eficiente

- ▶ Usa uma pilha para armazenar as runs
- ▶ Compara as ultimas runs da pilha
- ▶ Itercala sempre runs comparáveis no tamanho

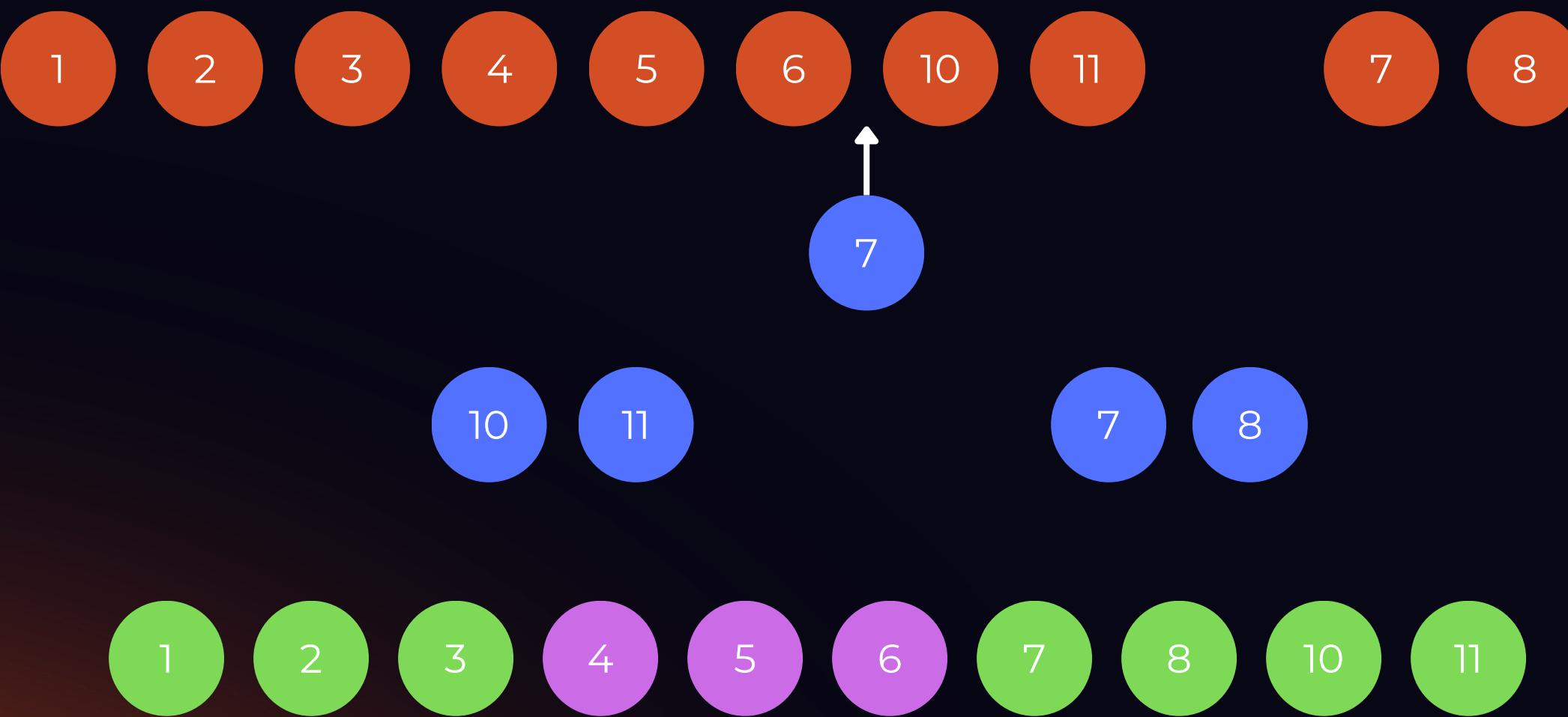


Entendendo o Algorítimo



Galloping Mode

Objetivo: Idêntica runs dominantes através do limiar e otimiza merge

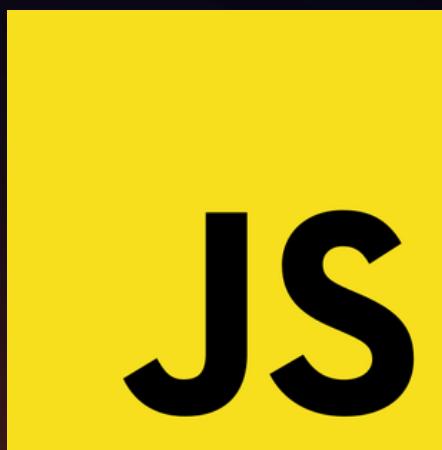


Análise de Desempenho



A análise levou em consideração duas métricas principais: tempo e gasto de memória. A avaliação foi estruturada em três eixos comparativos:

1. Comparação entre as linguagens: diferença de desempenho em C, C++, Java, JavaScript e Python
2. Estruturas de dados: como diferentes estruturas performam diante do mesma operação
3. Escalabilidade: desempenho em diferentes entradas de dados



Ferramentas de Medição

Tempo

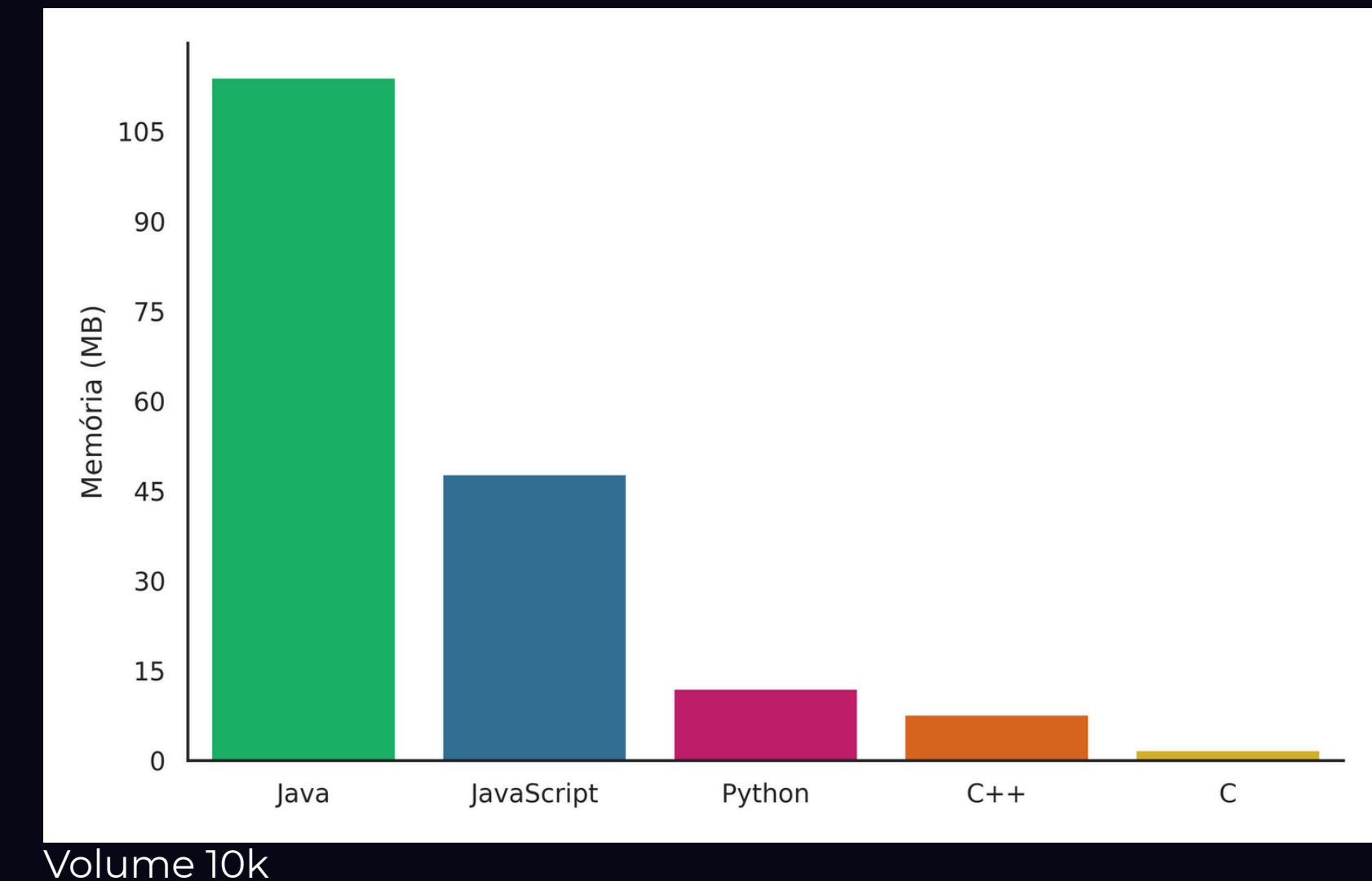
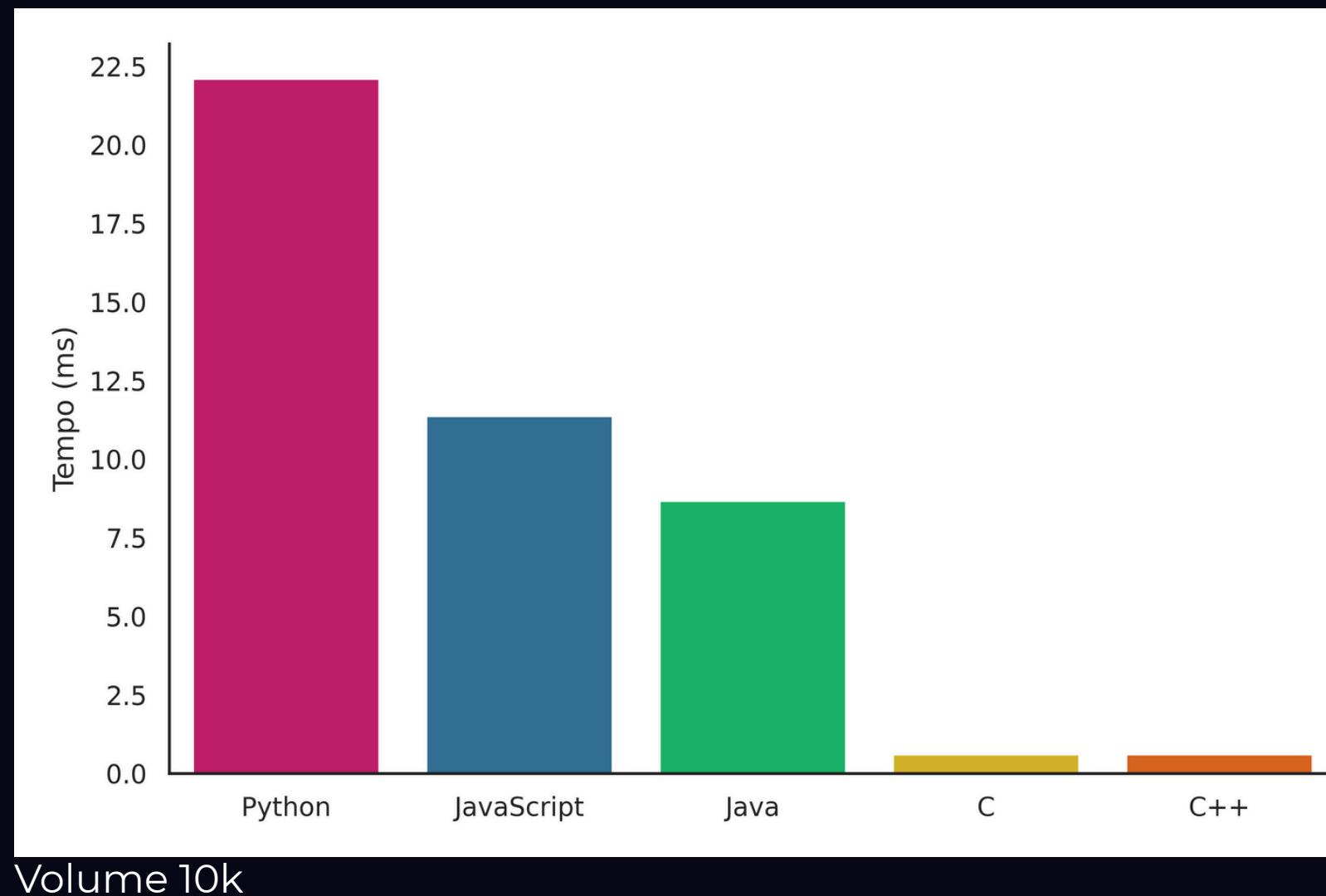
- 01** time.h em C
- 02** chrono em C++
- 03** System.nanoTime() em Java
- 04** time em Python
- 05** performance.now() em JavaScript

Memória

- 01** /usr/bin/time -v

Comparação entre linguagens

Sabe-se que a linguagem de programação utilizada na implementação de um algoritmo pode influenciar significativamente seu desempenho, especialmente em termos de tempo de execução e consumo de memória.



Comparação entre Linguagens

Analisando os gráficos, é significante a diferença de performance de C e C++, tanto em tempo de memória quanto em tempo de execução. Isso ocorre por 2 grandes motivos: modelo de execução e gerenciamento de memória

Compilação e execução

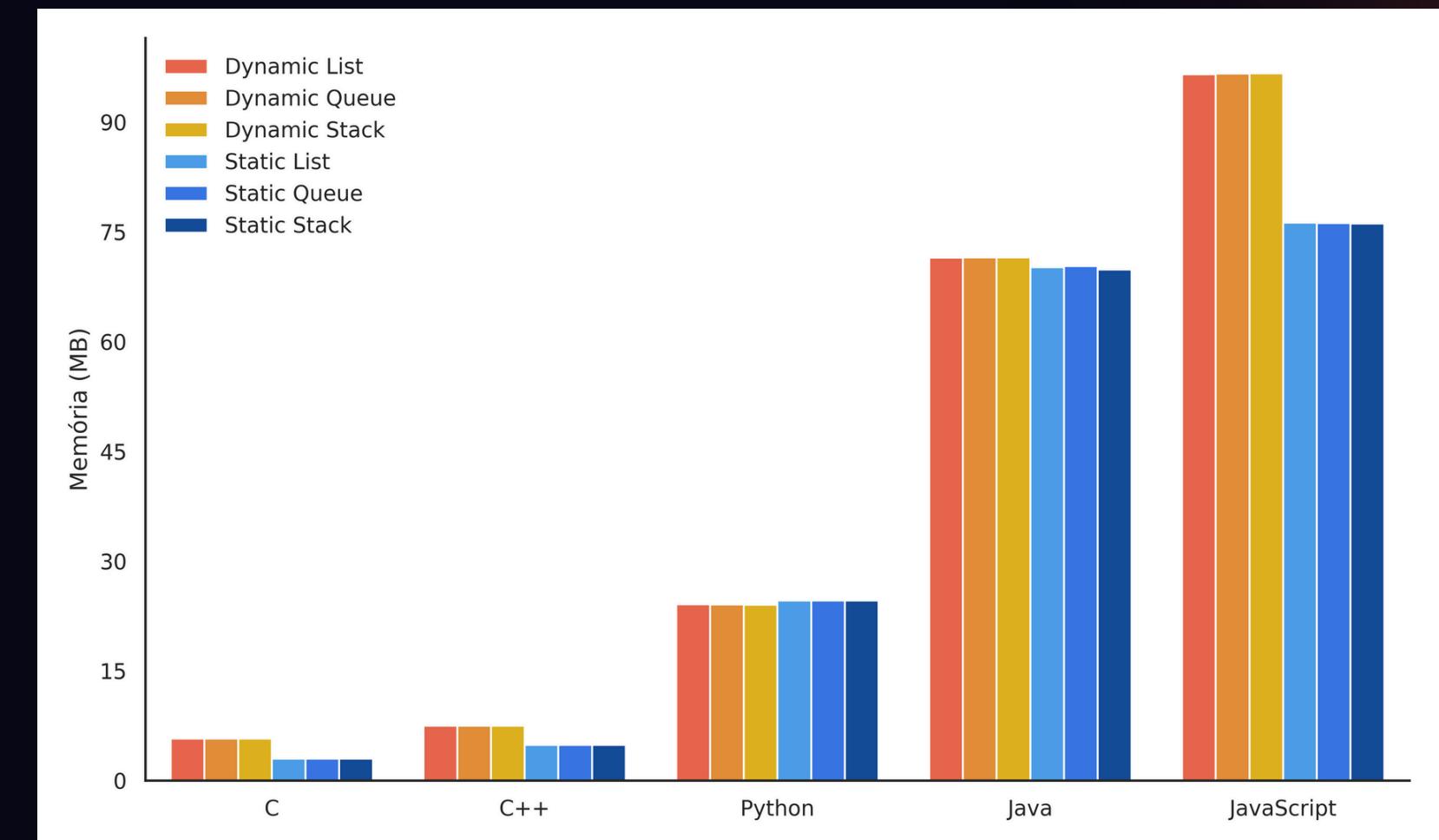
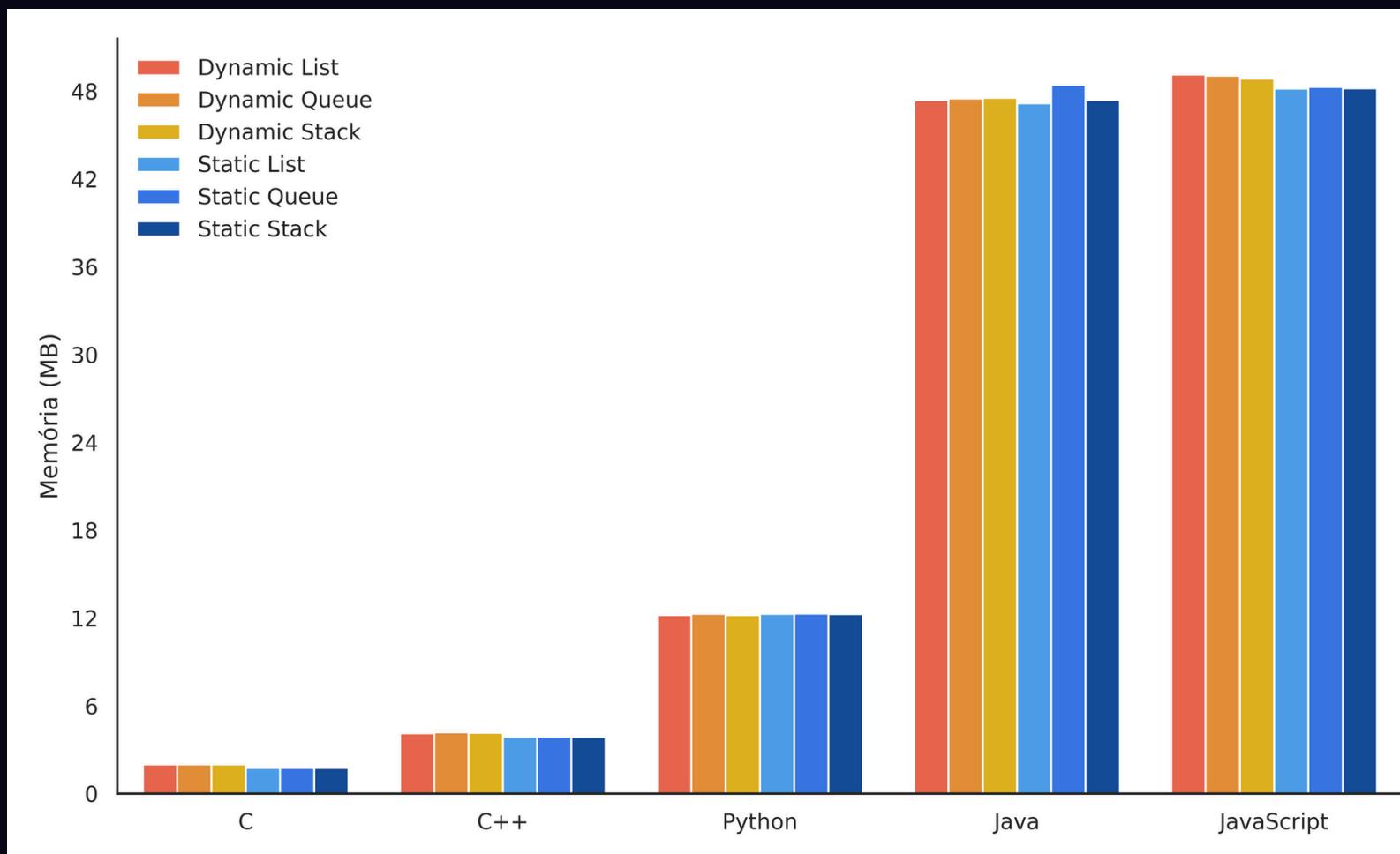
- C/C++
 - código nativo de máquina.
- Outras linguagens
 - interpretadas
 - máquinas virtuais

Gerenciamento de memória

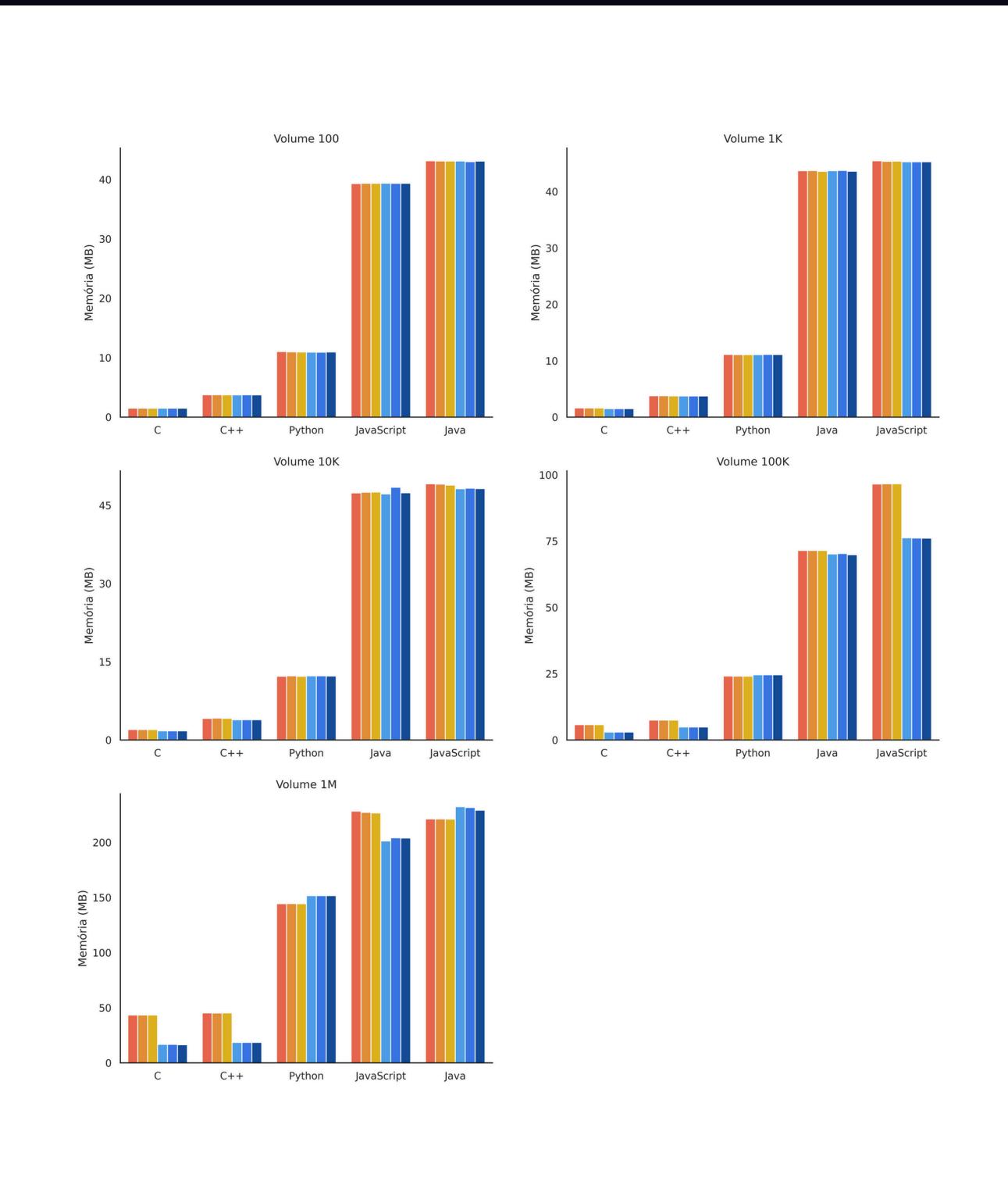
- C/C++
 - manipulação direta de memória
 - + erros
- Garbage Collector
 - + segurança
 - - velocidade
- Java e JavaScript operam em máquinas virtuais.

Comparação entre estruturas de dados

- Diferenças entre lista, fila e pilha foram pouco significativas.
- Estruturas estáticas tiveram desempenho ligeiramente superior.
- Ganhos em tempo e memória estão ligados à alocação contígua e melhor uso de cache.
- Linguagem de programação continua sendo o principal fator de desempenho.



Comparação entre volume de dados



- ▶ Testes com volumes entre 100 e 1 milhão de elementos.
- ▶ Diferença pequena em volumes baixos (até 1K).
- ▶ Estruturas dinâmicas sofrem com overhead de alocação e ponteiros.
- ▶ Impacto cresce em dados maiores (10K a 1M).
- ▶ Python teve os piores tempos, principalmente com estruturas dinâmicas.
- ▶ Java e JS melhoraram com dados maiores graças ao JIT.

Conclusão



Eficiência do algoritmo:

- ▶ Paradigma de programação como fator preponderante.
- ▶ Estrutura de dados é irrelevante (metodologia).
- ▶ C e C++ mais eficientes.
- ▶ Java, Python e JS menos eficientes.
- ▶ Aumento da diferença entre paradigmas (+volume)

Sugestão para novos estudos:

- ▶ Análise em volumes de dados maiores.
- ▶ Dados com características diferentes.
- ▶ Diferentes arquiteturas.

Obrigado pela atenção!

Duvidas?

