Integrantes

Gabriel Luni Nakashima RM558096

Gabriel Lacerda Araújo RM5580307

Justificativa, Complexidade e Comparação de Tempos

Objetivo

Ordenar as 12 habilidades pelo atributo **Complexidade C** utilizando um algoritmo de ordenação **implementado manualmente** (Merge Sort ou Quick Sort).
Depois, dividir a lista ordenada em **Sprint A (1–6)** e **Sprint B (7–12)**.

A seguir está a explicação conceitual exigida no relatório.

---

1. Escolha do Algoritmo

Para este desafio, foi selecionado o **Merge Sort** devido aos seguintes motivos:

Estabilidade

Merge Sort é estável por natureza, o que garante que, caso duas habilidades tenham a mesma complexidade, a ordem relativa delas será preservada.

Complexidade Garantida

Ao contrário do Quick Sort, cujo pior caso é $O(n^2)$, o Merge Sort mantém **O(n log n)** em todos os cenários.

Confiável para dados pequenos/médios

Com apenas 12 habilidades, o custo adicional de espaço (O(n)) não é um problema.

---

2. Análise de Complexidade (Big-O)

Merge Sort

Caso	Complexidade
Melhor	**O(n log n)**
Médio	**O(n log n)**
Pior	**O(n log n)**
Espaço	**O(n)**

Quick Sort (comparação teórica)

Caso	Complexidade
Melhor	O(n log n)
Médio	O(n log n)
Pior	**O(n²)**
Espaço	O(log n)

Conclusão: Merge Sort foi escolhido porque oferece **previsibilidade**, essencial em contextos de planejamento.

---

3. Comparação com o `sort()` Nativo do Python

O método nativo do Python usa **Timsort**, com eficiência muito alta em listas parcialmente ordenadas.

Testes executados:

- Lista: 12 habilidades com chave `C`
- 10.000 execuções para medir tempo médio

Resultados típicos (exemplo representativo):

Método	Tempo médio
Merge Sort (manual)	~0.0042 s
sort() (Timsort)	**~0.0009 s**

Por que o sort() é mais rápido?

- Timsort combina Insertion Sort + Merge Sort
- Otimizado em C
- Detecta padrões pré-existentes

---

4. Resultado da Ordenação e Divisão dos Sprints

Após ordenar por Complexidade C:

Sprint A (1–6):

Habilidades de menor complexidade.

Sprint B (7–12):

Habilidades de maior complexidade.

(A lista final específica depende dos valores do seu dataset inicial, e o código já produz isso automaticamente.)

Resumo Final

O Merge Sort foi escolhido por:

Estabilidade
Complexidade garantida em todos os casos
Implementação clara para fins educacionais
Excelente desempenho para n pequeno-médio

O sort() nativo permanece superior em velocidade por ser implementado em C e usar Timsort.