Trabalho 1 - Tipos Abstratos de Dados

- **Aluno:** Paulo Victor Fernandes de Melo
- **Aluno:** João Luiz Rodrigues da Silva
- **Aluna:** Rebecca Aimée Lima de Lima
- **Aluna:** Beatriz Jacaúna Martins

1. Busca em vetores e listas encadeadas

No código, desenvolvemos dois tipos abstratos de dados: um `vetor_t` e um `lse_t`, ambos representando um vetor de inteiros e uma lista encadeada de inteiros, respectivamente. Para ambos, foram implementadas funções para **inserir dados aleatórios**, **imprimir valores armazenados** e **buscar valores sequencialmente**. Para o `vetor_t`, foram implementadas também funções para **gerar um vetor ordenado** e **realizar a busca binária**.

No código, a função `questao1` executa a busca sequencial e binária no TAD `vetor_t`, retornando as medidas de tempo de execução para cada rodada, assim como a média e desvio padrão. Similarmente, a função `questao2` executa apenas a busca sequencial no TAD `lse_t`, retornando os resultados.

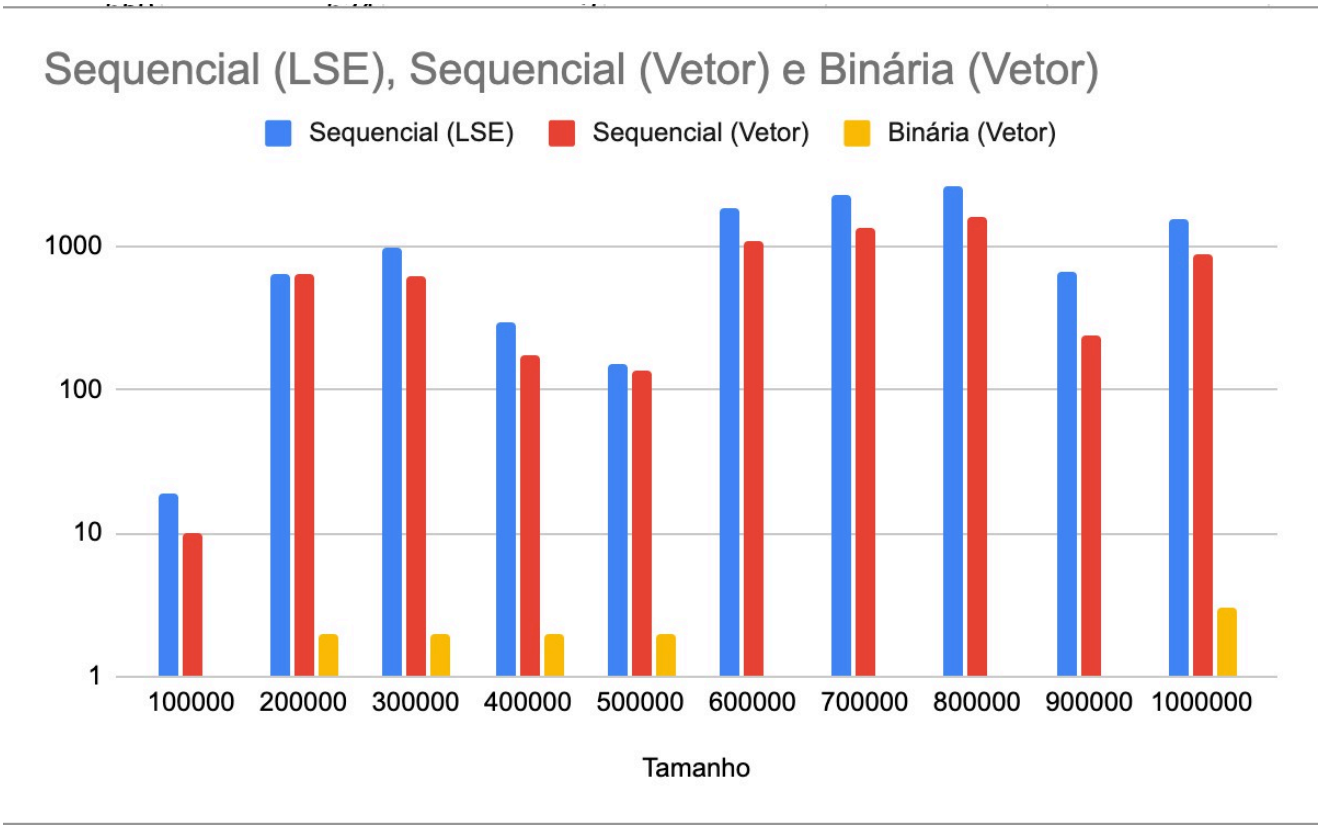
Durante a execução do código, a semente aleatória inicialmente inserida na função `srand` é armazenada e re-utilizada durante a geração aleatória e busca na lista encadeada. Desta forma, as listas encadeadas do teste possuirão os mesmos dados que os vetores, e os dados buscados nas listas também serão os mesmos. Assim, podemos comparar diretamente os tempos de execução da busca sequencial nos dois TADs.

Abaixo está uma tabela contendo os tempos de execução (em microssegundos[1]), média e desvio padrão da *busca sequencial* e *binária* em 30 vetores e listas encadeadas. Perceba como a busca binária, em comparação com a sequencial, possui um tempo de execução drasticamente menor, na casa dos **microssegundos**. Observe também como a busca na lista tende a demorar ≈ 2.1x mais que a busca sequencial.

	Busca seq. em vetor	Busca bin. em vetor	Busca seq. em lista
Iteração 1	2434,0	2,0	3862,0
Iteração 2	3289,0	3,0	6741,0
Iteração 3	2663,0	3,0	4283,0
Iteração 4	3330,0	3,0	6953,0
Iteração 5	3301,0	2,0	5924,0
Iteração 6	404,0	4,0	955,0

	Busca seq. em vetor	Busca bin. em vetor	Busca seq. em lista
Iteração 7	3311,0	2,0	5942,0
Iteração 8	2646,0	3,0	6322,0
Iteração 9	979,0	3,0	1766,0
Iteração 10	3332,0	3,0	9368,0
Iteração 11	3289,0	3,0	7219,0
Iteração 12	1695,0	3,0	5450,0
Iteração 13	2249,0	3,0	4453,0
Iteração 14	2150,0	3,0	5109,0
Iteração 15	640,0	7,0	1144,0
Iteração 16	3324,0	3,0	7778,0
Iteração 17	2350,0	0,0	4343,0
Iteração 18	887,0	3,0	2152,0
Iteração 19	3301,0	3,0	6141,0
Iteração 20	3423,0	3,0	8520,0
Iteração 21	1455,0	3,0	2726,0
Iteração 22	3330,0	3,0	7939,0
Iteração 23	2679,0	3,0	4864,0
Iteração 24	1708,0	3,0	4033,0
Iteração 25	3356,0	3,0	6053,0
Iteração 26	1681,0	3,0	3949,0
Iteração 27	3327,0	2,0	6048,0
Iteração 28	3343,0	3,0	7806,0
Iteração 29	320,0	2,0	623,0
Iteração 30	3347,0	3,0	7731,0
Media	2451,433333	2,9	5206,566667
Desvio padrão	1006,837282	1,011599	2306,412693

Variamos os tamanhos dos vetores, de **cem em cem mil** até **um milhão**, colocando os resultados das execuções em um gráfico em escala logarítmica. Os tempos de execução também estão em microssegundos[1-1].



Vemos no gráfico que os tempos de execução tendem a aumentar junto com o tamanho do vetor, apesar dos erros em alguns testes devido a fatores externos. Percebe-se também que, como visto na tabela, a busca binária quase não aparece no gráfico, devido ao seu tempo de execução na ordem dos microssegundos, e a busca sequencial na lista encadeada sempre parece demorar um pouco mais que no vetor.

2. Ordenação de Vetores

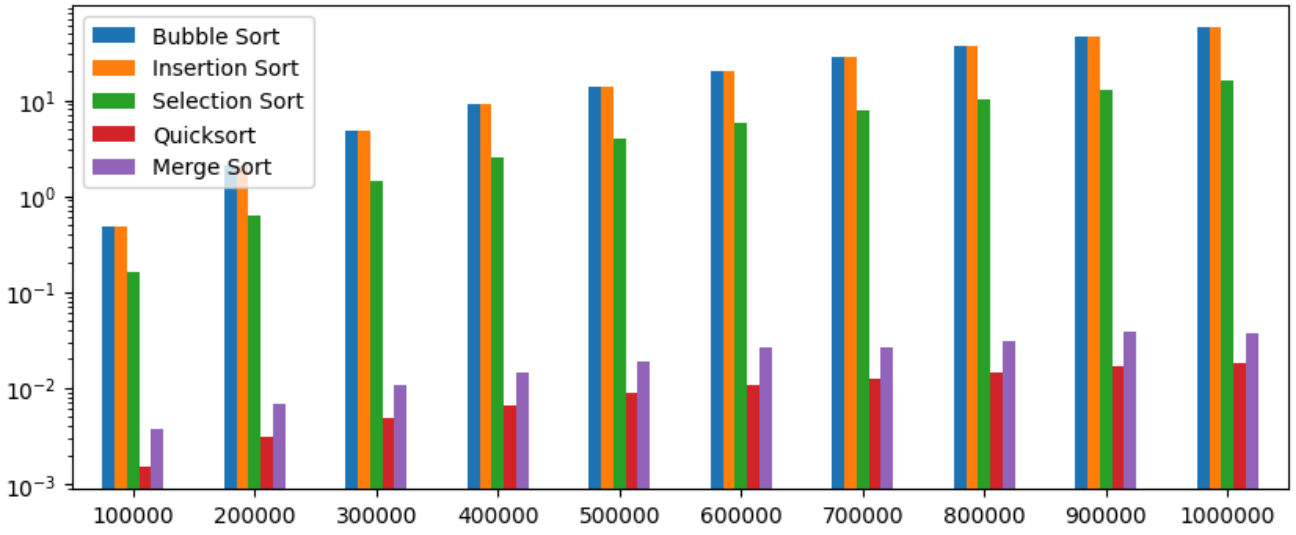
A implementação do TAD `vetor_t` desenvolvida anteriormente foi expandida com funções para ordenar o vetor usando os seguintes métodos: **bubble sort**, **insertion sort**, **selection sort**, **quicksort** e **merge sort**. Também foi implementada uma função de utilidade para **copiar** um objeto do tipo `vetor_t` para outro do mesmo tamanho.

No código, a função `questao3` executa cada um dos 5 algorítimos de ordenação em **10 vetores diferentes**, com 100 mil elementos cada, usando o mesmo vetor para cada algorítimo. A função de cópia previamente mencionada garante que os vetores aleatórios desordenados não são perdidos após a execução de um algorítimo de ordenação.

Abaixo está uma tabela contendo os tempos de execução (em segundos), média e desvio padrão para cada algorítimo. Perceba que o *bubble sort* possui um tempo de execução na ordem de minutos quando ordenando um vetor de 100 mil elementos. Por outro lado, o *quicksort* e o *merge sort* possuíram tempo de execução na ordem dos milissegundos, demonstrando-os adequados para uso em ordenação de vetores grandes.

	Bubble sort	Insertion sort	Selection sort	Quicksort	Merge sort
Iteração 1	55,753	8,776	15,911	0,017	0,027
Iteração 2	55,185	8,789	15,891	0,018	0,027
Iteração 3	54,792	8,788	15,836	0,017	0,027
Iteração 4	54,876	9,036	15,943	0,018	0,028
Iteração 5	55,014	8,821	15,916	0,018	0,027
Iteração 6	55,005	8,792	15,956	0,017	0,027
Iteração 7	54,826	8,814	16,046	0,018	0,027
Iteração 8	55,060	8,872	16,027	0,018	0,027
Iteração 9	54,919	8,781	15,870	0,018	0,027
Iteração 10	55,343	8,796	15,893	0,018	0,027
Media	55,077	8,827	15,929	0,018	0,027
Desvio padrão	0,275399	0,074567	0,062855	0,000233	0,000195

Similarmente à questão de busca, fizemos um gráfico em escala logarítmica mostrando como o tempo varia de acordo com o tamanho do vetor, variando de **dez em dez mil** até **cem mil** elementos. Perceba como, na ordenação, não ocorreram tantas grandes variações no tempo de execução quanto na busca. Isto provavelmente se deve ao fato da busca ser mais afetada por **valores aleatórios de busca**[2] quando comparada com a ordenação.



---

1. Note que $1 \text{ segundo} = 10^6 \ \mu s$. ↩ ↩
2. Em cada rodada das buscas de tamanhos diferentes, os valores sendo buscados eram diferentes, podendo estar fora dos vetores/listas. Na ordenação, ao aumentar o tamanho dos vetores, a parte

anterior do vetor era gerada com **a mesma semente aleatória**, havendo menos variação. ↩