Universidade Tecnológica Federal do Paraná
Bacharelado em Ciências da Computação

Trabalho Prático: Uma análise de algoritmos clássicos

Estudantes:
Pedro Itiro Nagao
João Ricardo Zulato Reberti

Professor: Rodrigo Hübner

Campo Mourão
2025

Introdução

O presente trabalho demonstra e analisa algoritmos de ordenação e busca clássicos. Que são:

- Selection Sort (original e otimizado);
- Bubble Sort (original e otimizado);
- Insertion Sort;
- Busca binária;
- Busca Sequencial.

Todos os algoritmos foram implementados manualmente pelos membros do grupo usando a linguagem de programação *C++* no padrão 17 e compilados com o *Cmake*.

Características dos algoritmos

Selection Sort

Itera a coleção para conseguir o menor elemento. Depois, troca este valor com o que se encontra na 1° posição da coleção.
Assim, o processo continua para elementos subsequentes até que a lista esteja ordenada.
É o algoritmo de ordenação mais simples, porém sua complexidade temporal média é de $O(n^2)$. O que pode ser lento para listas muito grandes.

A versão otimizada implementada no código-fonte itera a coleção em ambos os lados ao mesmo tempo. Da direita para a esquerda, se procura pelo menor elemento; do lado oposto, se procura pelo maior elemento.
Quando o maior e menor elemento são encontrados, o menor é colocado no início e o maior no final. Isso reduz as iterações significativamente.

Bubble Sort

Outro algoritmo com lógica simples, onde se troca elementos adjacentes que não estão ordenados (isto é, o primeiro ser menor que o segundo) repetidas vezes até a coleção estar totalmente ordenada, que é $O(n^2)$.

É também custoso como o *Selection Sort*. A única mudança notável que pode ser implementada sem comprometer a natureza da ordenação é checar se a estrutura de dados está ordenada ao analisar se houve alguma troca durante as iterações. Se em algum momento não houve trocas, então o vetor está ordenado.

Insertion Sort

Uma analogia comum a este algoritmo é que este “ordenasse baralho”, onde colocamos cartas menores à direita até que o baralho esteja ordenado.
Assume-se que o primeiro esteja ordenado e, a partir disso, o primeiro elemento que for menor ou igual que nosso primeiro número é inserido antes deste. Para os elementos subsequentes, se itera até o início novamente até que uma posição onde o possível anterior seja menor que o nosso atual além do possível próximo seja maior ou igual. Colocamos o elemento ali.
Consegue ser o mais rápido dentre os algoritmos citados anteriormente. Porém é ainda $O(N^2)$.

Busca Binária

Este algoritmo exige que a estrutura de dados esteja ordenada. Ele começa no centro da coleção.

- Se o elemento nesta posição for igual ao que se procura: temos a posição dele;
- Se o elemento aqui for maior que o alvo: então devemos procurar mais à frente;
- Analogamente, se o elemento central for menor, então procuramos mais atrás.

Assim, corta-se a coleção em dois vetores menores e repete-se o processo anterior até achar o elemento desejado e sua posição.
Tem complexidade temporal média de $O(\log n)$. O que é rápido especialmente para estruturas grandes.

Busca Sequencial

É o algoritmo de busca mais simples. Percorre-se do início ao final da coleção comparando cada elemento com o alvo até que se ache o primeiro que seja igual ao que se procura.
Apesar de não ser complexo, a busca sequencial costuma fazer muitas comparações.

Ambiente de teste

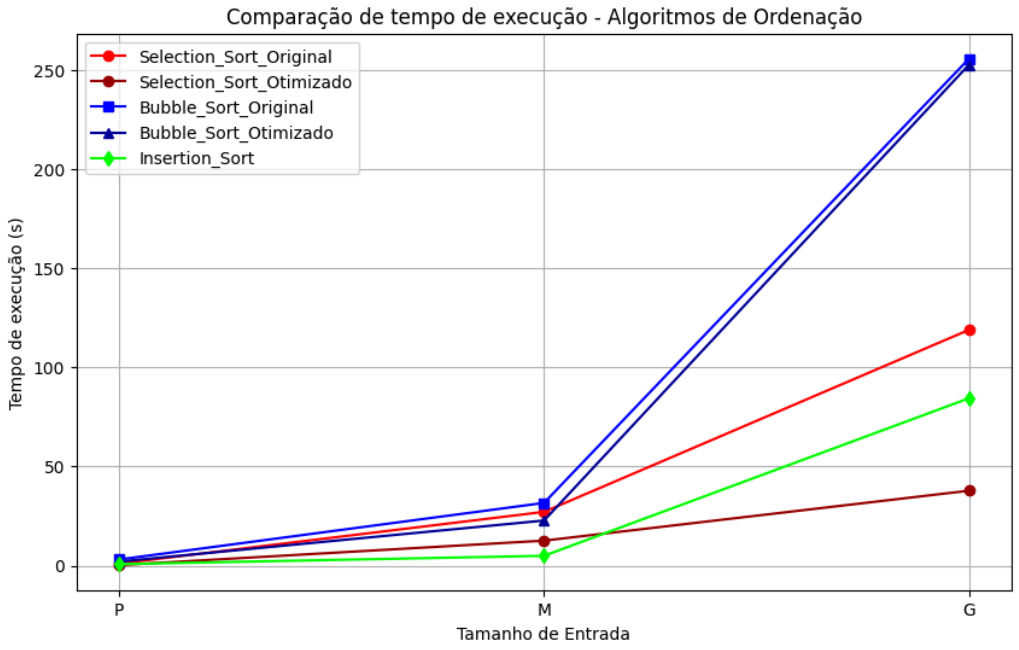
Todos os dados foram criados usando números aleatórios de uma distribuição normal padrão tendo como limites o maior e o menor número possível de um inteiro com sinal.
Foram criadas 3 coleções em arquivos binários que são criados e lidos na execução:

- Para uma coleção pequena, usamos 24.000 números;
- Para uma coleção média, 120.000 números;
- Para uma coleção grande, temos 250.000 número;

Para os algoritmos de busca, é eleito um número aleatório na execução antes de executar os algoritmos.
Para a execução dos testes, foi utilizado um *notebook* com as seguintes especificações:

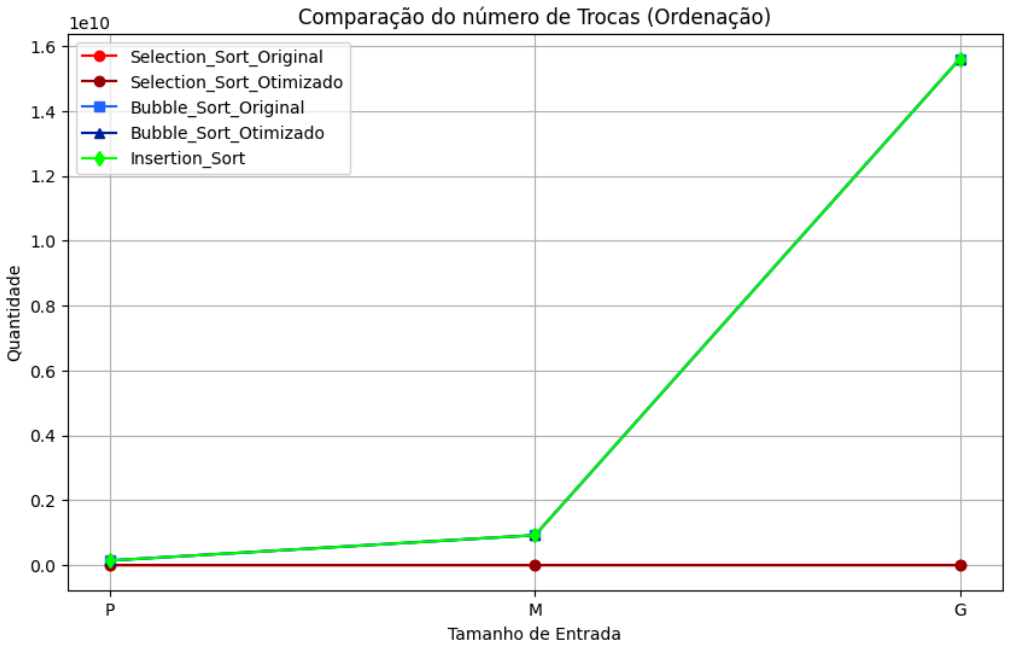
- CPU: AMD Ryzen 7 5700U (16) @ 4.37 GHz;
- RAM: 20 GB em DDR4 3600mhz;
- SIstema Operacional: Arch Linux com Kernel 6.14.9-arch 1-1;
- Compilador C++: GCC 15.1.1.

Análise dos resultados



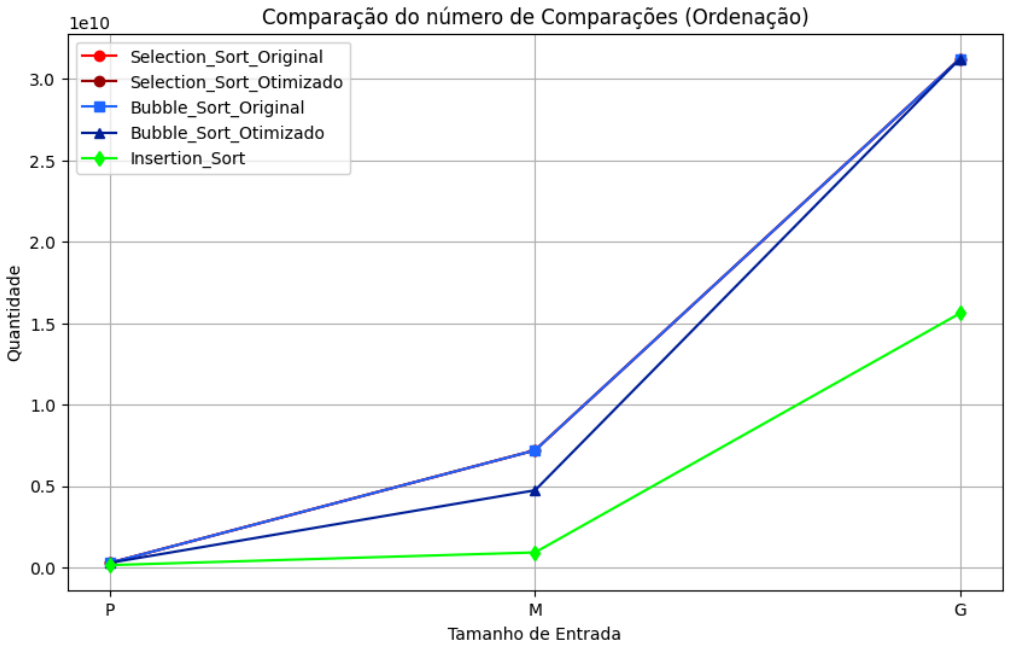
Algoritmo	Tempo decorrido para cada coleção em Segundos		
	P(pequeno)	M(médio)	G(grande)
Selection Sort Padrão	1.05036	27.1239	118.878
Selection Sort Otimizado	0.279954	12.6153	37.7836
Bubble Sort Padrão	3.11797	31.6457	255.561
Bubble Sort Otimizado	2.06691	22.7581	252.553
Insertion Sort	0.763719	4.97227	84.3878

O comportamento dos algoritmos segue o esperado teoricamente. A única surpresa aqui é o i*nsertion sort* ter um tempo maior que o *selection sort* otimizado. Isso provavelmente se deve às inserções



Algoritmo	Trocas feitas em cada coleção		
	P(pequeno)	M(médio)	G(grande)
Selection Sort Padrão	23.982	119.990	249.987
Selection Sort Otimizado	23.982	105.730	249.989
Bubble Sort Padrão	142.953.346	922.597.920	15.611.574.079
Bubble Sort Otimizado	142.953.346	922.597.920	15.611.574.079
Insertion Sort	142.953.346	922.597.920	15.611.574.079

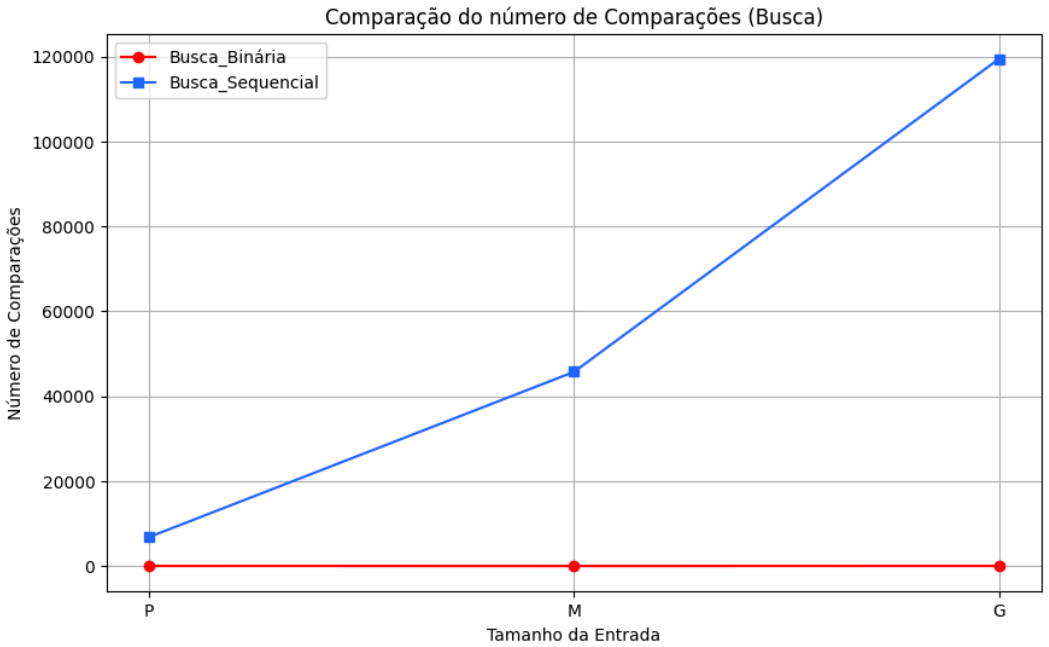
Novamente, o comportamento segue a teoria, como todos os algoritmos, exceto os *selection 's sort,* executam trocas excessivas, números altos é esperado.
O *selection sort* otimizado consegue ser mais eficiente devido a ter menos iterações ao procurar pelo menor a maior número ao mesmo tempo e evitar posições já passadas.



Algoritmo	Comparações feitas em coleção		
	P(pequeno)	M(médio)	G(grande)
Selection Sort Padrão	287.988.000	7.199.940.000	31.249.875.000
Selection Sort Otimizado	287.895.716	7.199.561.655	31.248.628.016
Bubble Sort Padrão	287.988.000	7.199.940.000	31.249.875.000
Bubble Sort Otimizado	287.978.130	4.739.884.704	31.249.026.747
Insertion Sort	142.977.333	922.717.910	15.611.824.069

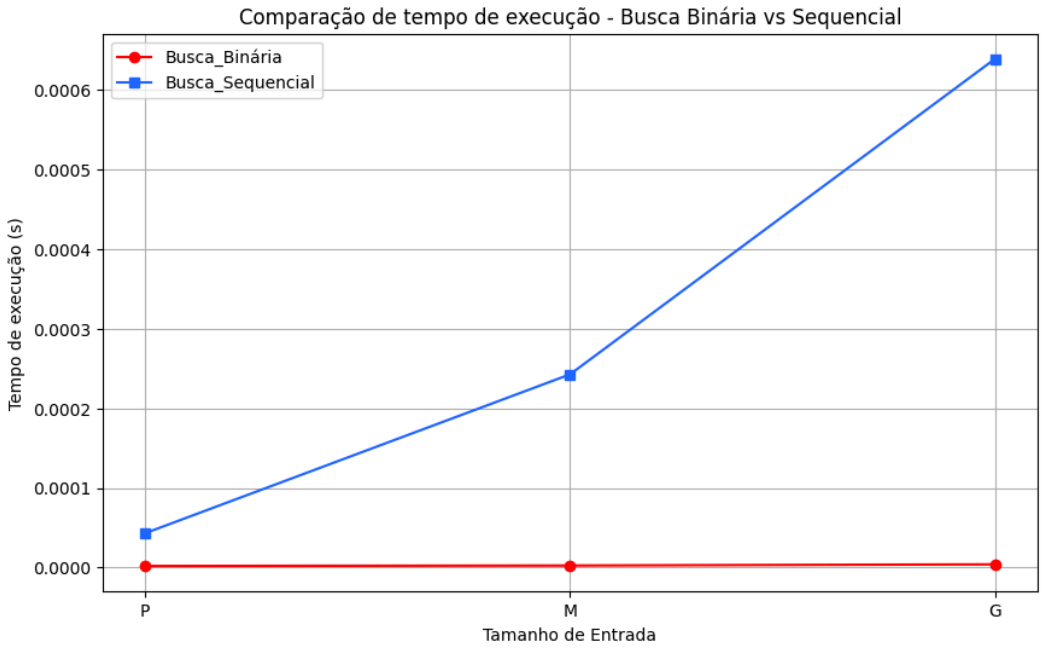
Os algoritmos *selection sort* e *bubble sort* originais ficam praticamente empatados em todas as coleções com o *bubble sort* sendo um número um pouco maior. Enquanto que suas versões otimizadas conseguem fazer uma quantidade menor de comparações.
Além disso, o *insertion sort* consegue a menor quantidade de comparações

Para os algoritmos de busca, obtivemos os seguintes resultados:



Algoritmo	Comparações feitas em cada coleção		
	P(pequeno)	M(médio)	G(grande)
Busca Sequencial	13	1	18
Busca Binária	6.822	45.733	119.449

Aqui, o algoritmo de busca binária consegue um desempenho muito superior ao fazer menos comparações que a busca sequencial.



Algoritmo	Tempo decorrido em cada coleção em Milisegundos		
	P(pequeno)	M(médio)	G(grande)
Busca Sequencial	1.886×10^{-3}	2.444×10^{-3}	3.912×10^{-3}
Busca Binária	4.309×10^{-2}	0.242769	0.639257

Devido aos “saltos” que a busca binária faz, seu tempo consegue ser muito menor comparado à busca sequencial.

Conclusão

- **Selection Sort Original vs Otimizado**
  A implementação otimizada (busca simultânea de mínimo e máximo) reduziu pela metade o número de passadas externas, resultando em tempos ~2× menores que a versão clássica em vetores aleatórios grandes, sem alterar a ordem assintótica (ambos $O(n^2)$).

- **Bubble Sort Original vs Otimizado**
  Checar se houve alguma troca interrompe o laço externo quando não há trocas, mas em entradas completamente aleatórias isso quase não ocorre, de modo que o ganho prático foi mínimo em vetores grandes.

- **Insertion Sort**
  Fez ~$n(n-1)/4$ comparações e deslocamentos em média, coerente com a teoria, mas sofreu penalidade de escrita de memória em vetores aleatórios, ficando mais lento que o Selection Otimizado, embora realize menos comparações. Em vetores quase ordenados, porém, cairia a $O(n)$.

- **Busca Binária vs Sequencial**
  A busca binária usou ~$\log_2 n$ comparações (≈18 para 250 000) e foi microssegundos mais rápida que a busca sequencial (~110 000 comparações).

Em resumo, os resultados confirmam as previsões teóricas e mostram que, em entradas grandes e aleatórias, o **Selection Sort Otimizado** supera o Insertion Sort e o Bubble Sort Otimizado, enquanto a **Busca Binária** mantém sua superioridade sobre a busca linear.