FAETERJ - Paracambi
Curso: Análise e Desenvolvimento de Sistemas
Disciplina: Estrutura de Dados Prof: Carlos Eduardo

Análise Comparativa dos Algoritmos de Ordenação: Selection Sort e Shell Sort

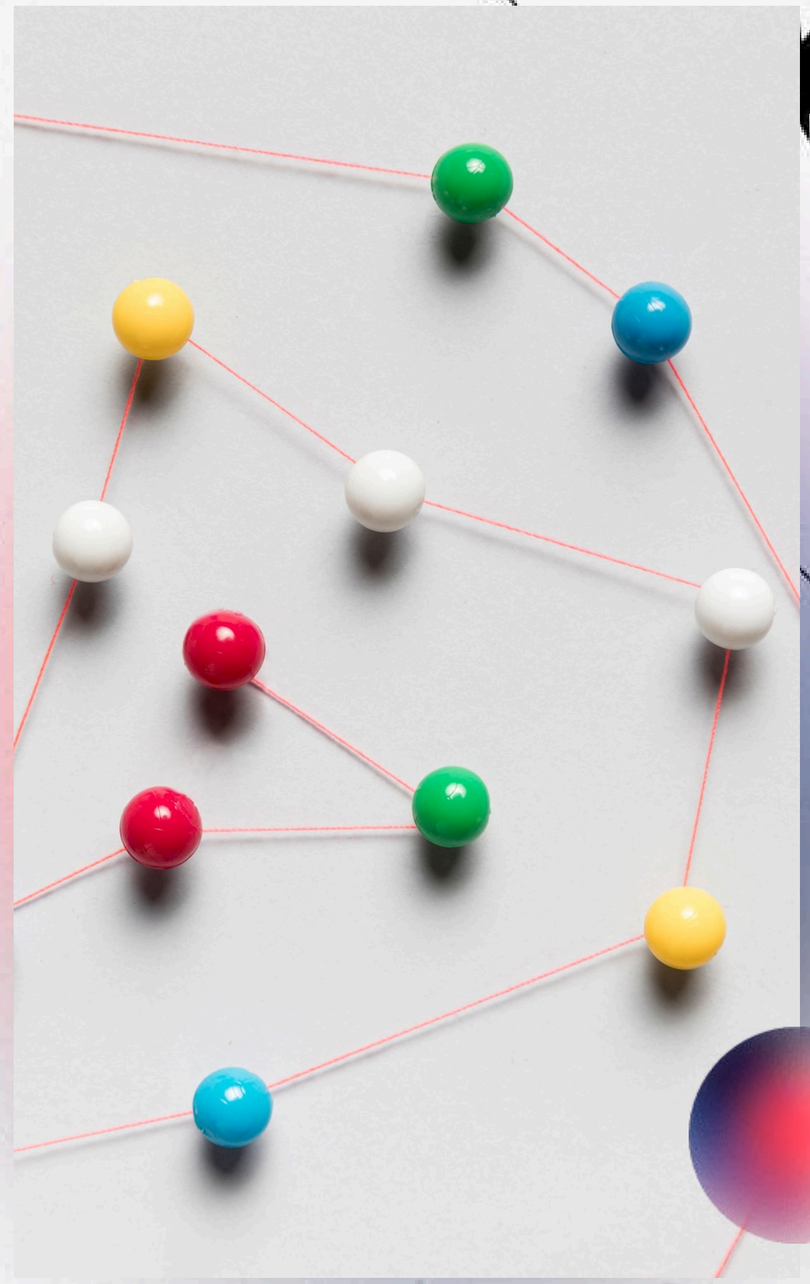
Alunos: Isabelle Gonçalves da Silva
Vinícius Aniceto dos Santos

Introdução ao problema de ordenação

O que é Ordenação? É o processo de organizar uma lista de elementos (números, palavras, objetos) de acordo com uma ordem específica (crescente, decrescente, alfabética, etc.).

Por que Ordenar?

- Busca eficiente: Encontrar um elemento específico em uma lista ordenada é muito mais rápido do que em uma lista desordenada.
- Análise de dados: Muitas técnicas de análise de dados exigem que os dados estejam ordenados.
- Organização de informações: A ordenação é fundamental para organizar informações de forma lógica e intuitiva para o usuário.

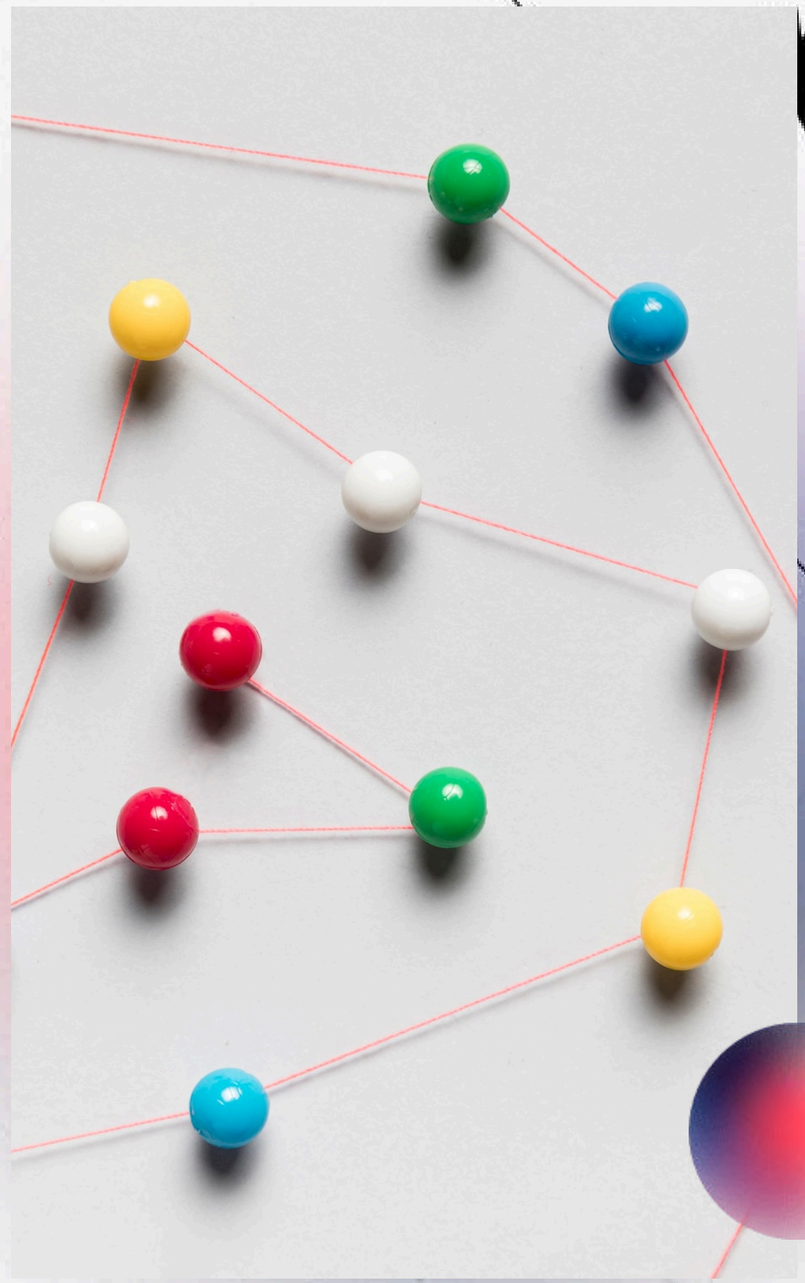


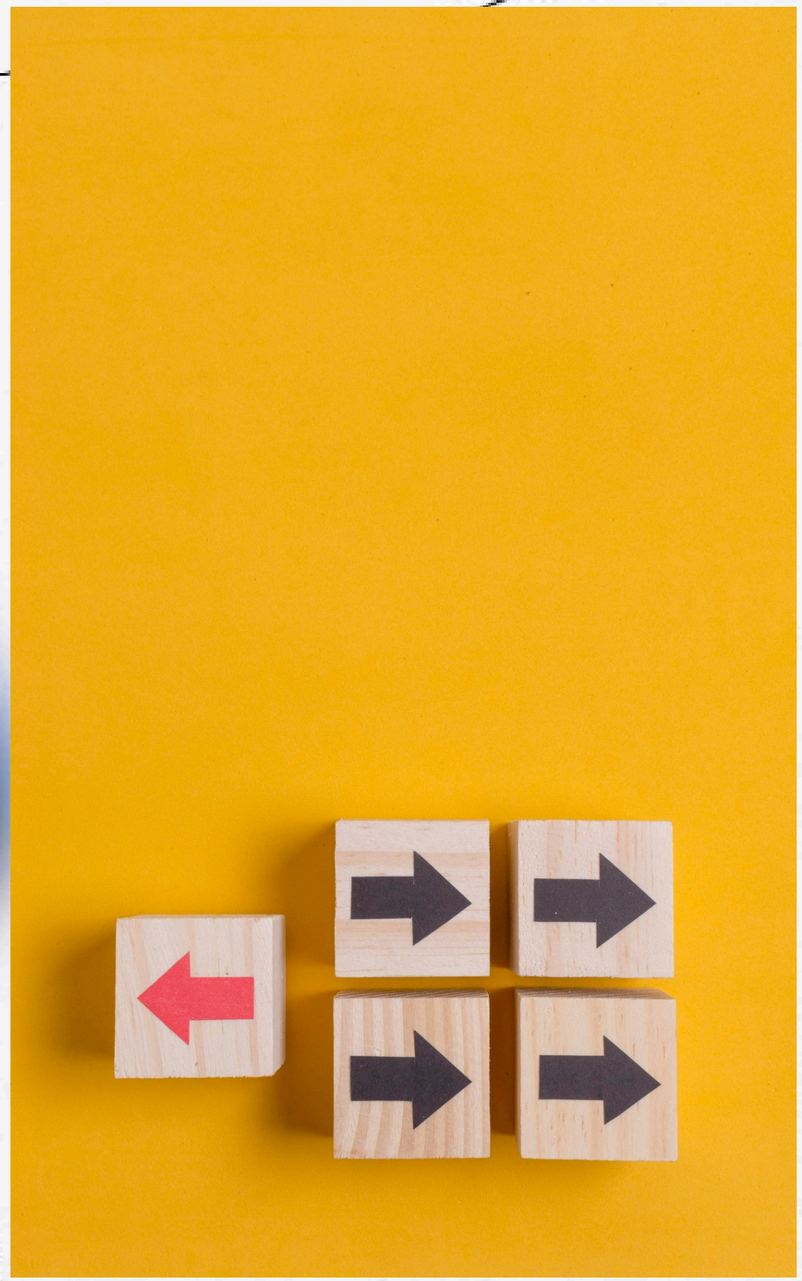
Aplicações do problema de ordenação

A ordenação tem inúmeras aplicações em diversas áreas, como:

- Bases de dados: Ordenar registros por um determinado campo (nome, data, valor).
- Sistemas operacionais: Ordenar processos, arquivos e diretórios.
- Compiladores: Ordenar símbolos e tabelas.
- Algoritmos de busca: A maioria dos algoritmos de busca binária exige que os dados estejam ordenados.
- Gráficos e visualizações: Ordenar dados para criar gráficos e visualizações mais informativas.
- Machine Learning: Ordenar dados para treinamento de modelos de aprendizado de máquina.

Em resumo, a ordenação é uma operação fundamental em computação, e a escolha do algoritmo de ordenação depende das características dos dados e dos requisitos da aplicação.





Descrição dos algoritmos escolhidos

- Selection Sort: Encontra o menor elemento da lista e o coloca na primeira posição. Repete o processo para o restante da lista, encontrando o segundo menor elemento, o terceiro menor, e assim por diante.

- Shell Sort: Uma variação do Insertion Sort, mas mais eficiente. Divide a lista em sublistas menores e ordena cada sublista. Em seguida, diminui o tamanho das sublistas e repete o processo até a lista inteira estar ordenada.

Pseudocódigo

```
Início
  Função Ordenacao_Selecao(V, N)
    PARA I DE 0 ATÉ N-1
      MENOR ← I
      PARA J DE I+1 ATÉ N
        SE V[J] < V[MENOR] ENTÃO
          MENOR ← J
        FIM SE
      FIM PARA
      AUX ← V[I]
      V[I] ← V[MENOR]
      V[MENOR] ← AUX
    FIM PARA
  FIM Função
Fim
```

```
Início
  Função ShellSort(V, N)
    gap ← N / 2
    ENQUANTO gap > 0 FAÇA
      PARA cada i de gap ATÉ N-1 FAÇA
        temp ← V[i]
        j ← i
        ENQUANTO j >= gap E V[j - gap] > temp FAÇA
          V[j] ← V[j - gap]
          j ← j - gap
        FIM ENQUANTO
        V[j] ← temp
      FIM PARA
      gap ← gap / 2
    FIM ENQUANTO
  FIM Função
Fim
```

Explicação de suas complexidades

A complexidade O(n2) descreve algoritmos cujo tempo de execução cresce proporcionalmente ao quadrado do tamanho da entrada. Isso significa que se a entrada dobra de tamanho, o tempo de execução quadruplica. Este tipo de complexidade é comum em operações onde cada elemento deve ser comparado ou operado com cada outro elemento.

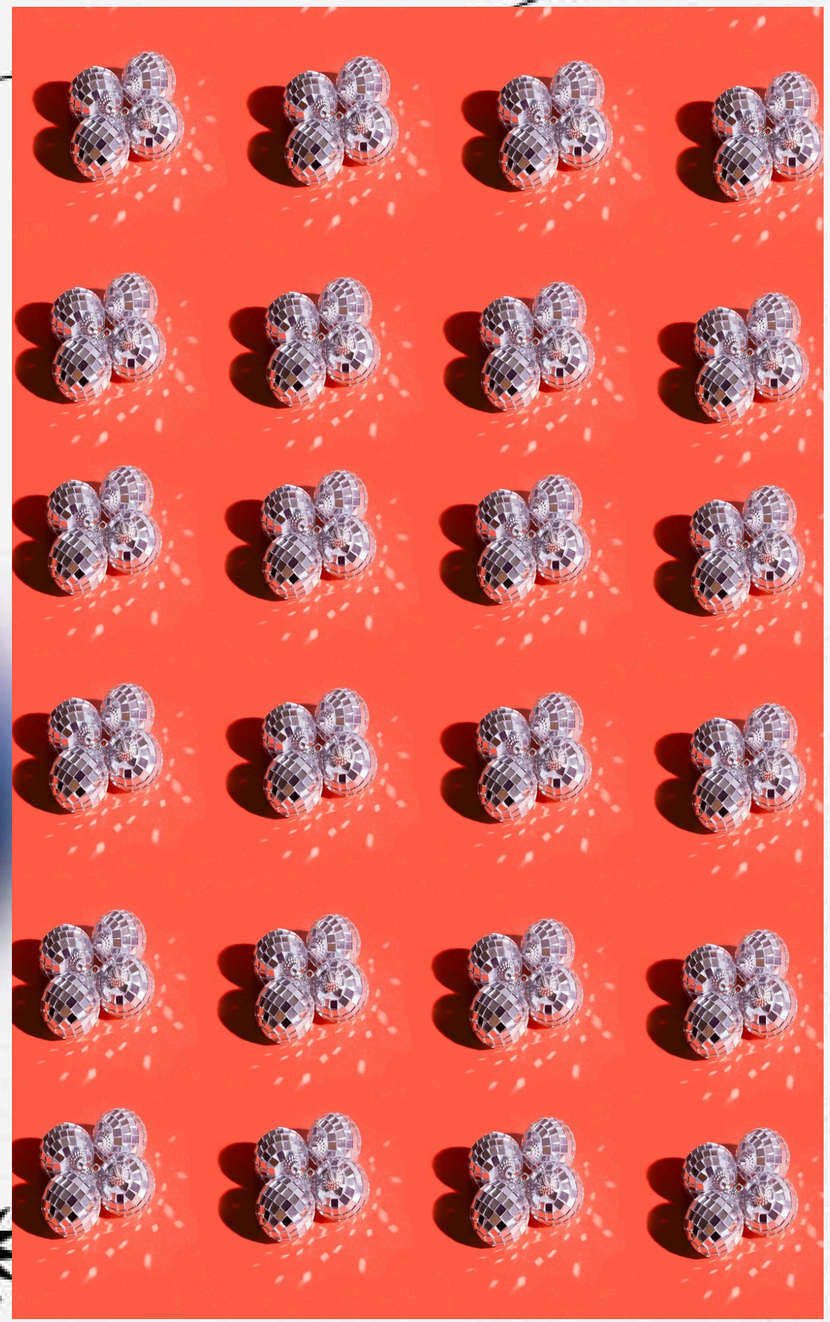
A complexidade O(n log n) descreve algoritmos cujo tempo de execução cresce proporcionalmente ao logaritmo do tamanho da entrada. Isso significa que se a entrada dobra de tamanho, o tempo de execução aumenta apenas por uma constante. Este comportamento é visto em algoritmos que dividem a entrada em partes menores a cada passo.

Complexidades

Crescimento do Tempo de Execução:
- O(n2): O tempo de execução aumenta muito rapidamente com o aumento do tamanho da entrada.
- O(n log n): O tempo de execução aumenta muito lentamente em comparação, mesmo com entradas muito grandes.

Escalabilidade:
- O(n2): Ineficiente para grandes entradas devido ao crescimento exponencial do tempo de execução. Apropriado para pequenas listas onde o desempenho não é uma grande preocupação.
- O(n log n): Muito eficiente e escalável, adequado para grandes entradas onde um crescimento lento do tempo de execução é crítico.

Utilização de Recursos:
- O(n2): Pode consumir uma quantidade substancial de tempo de CPU e memória quando a entrada é grande, devido ao número elevado de operações necessárias.
- O(n log n): Utiliza recursos de forma mais eficiente, com menos operações à medida que a entrada cresce.

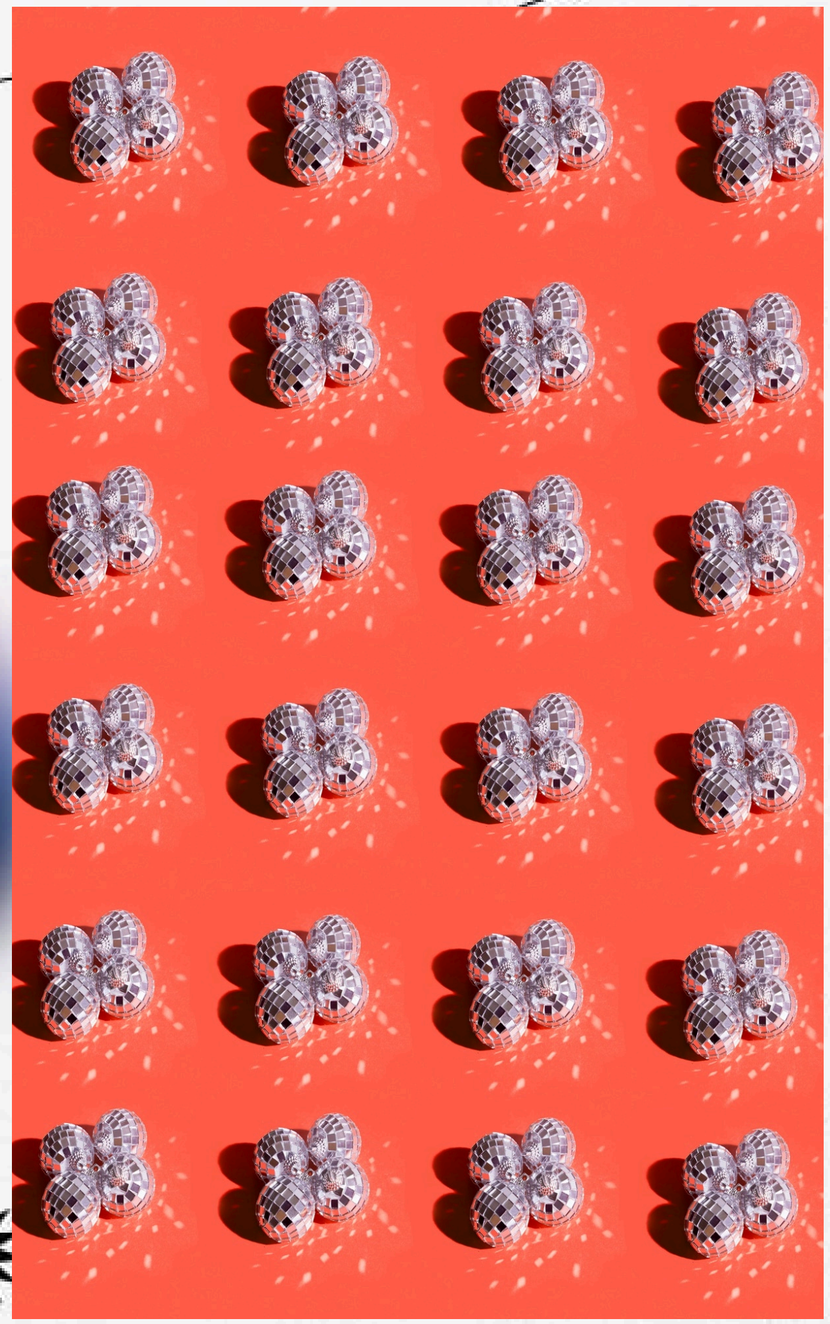


Descrição das instâncias de teste utilizadas

Os testes foram realizados com três categorias de tamanhos de dados:
- Dados Pequenos: 1.000, 2.000, 3.000, 4.000, 5.000.
- Dados Médios: 100.000, 200.000, 300.000, 400.000, 500.000.
- Dados Grandes: 1.000.000, 2.000.000, 3.000.000, 4.000.000, 5.000.000.

Os seguintes algoritmos foram utilizados para os testes:
- Selection Sort
- Shell Sort

Os dados foram divididos em duas categorias:
- Dados Aleatórios
- Dados em Sequência



Descrição do PC utilizado e do programa

GDB online

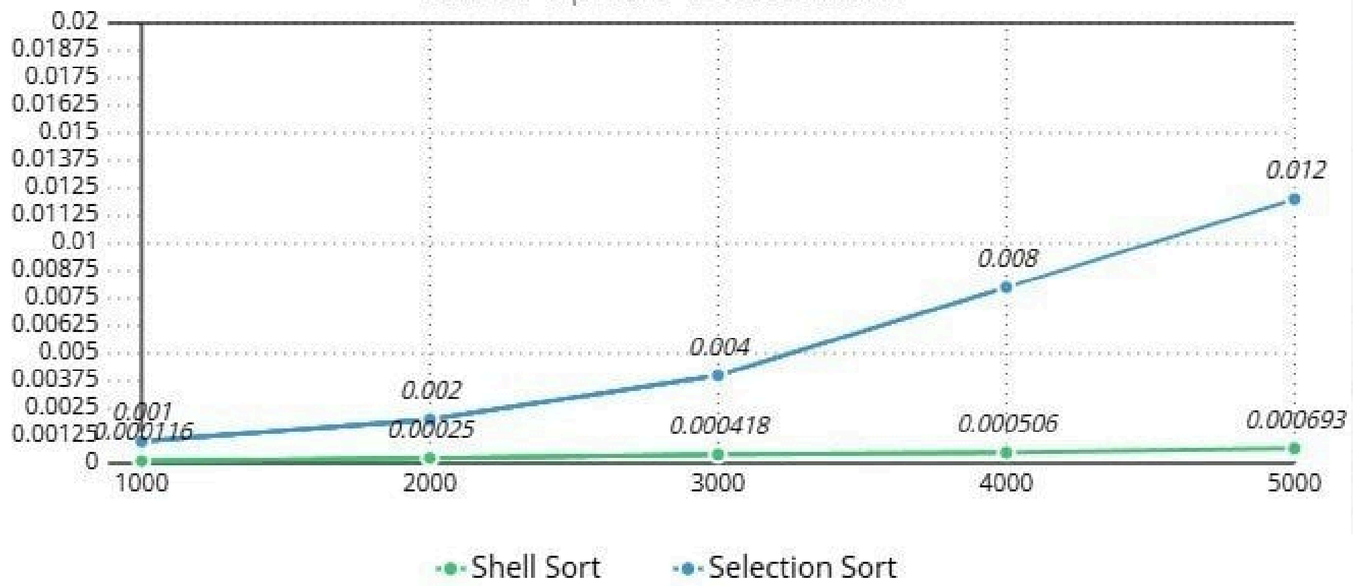
Processador: AMD Ryzen 5 5600G

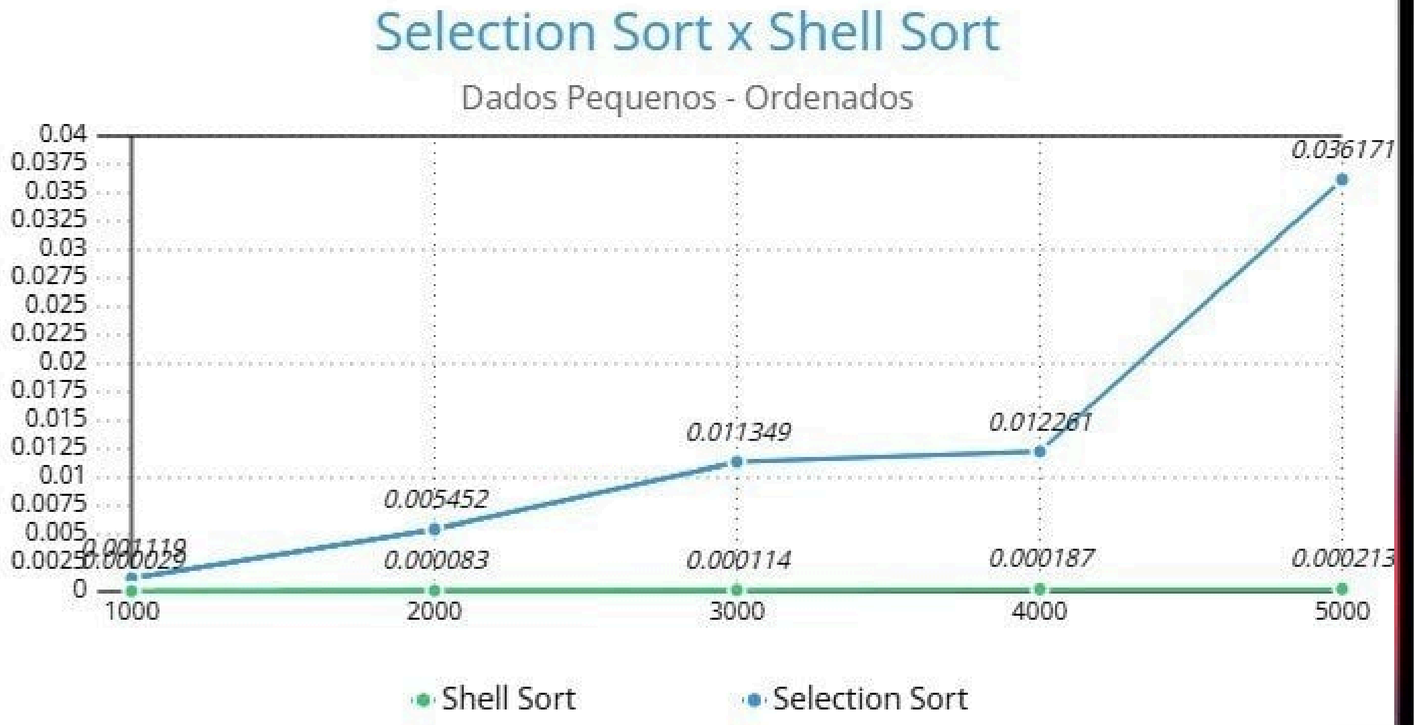
Placa Mae: A520M-HVS

Memória RAM: 2x6GB - DDR4. Frequência: 1600MHz.

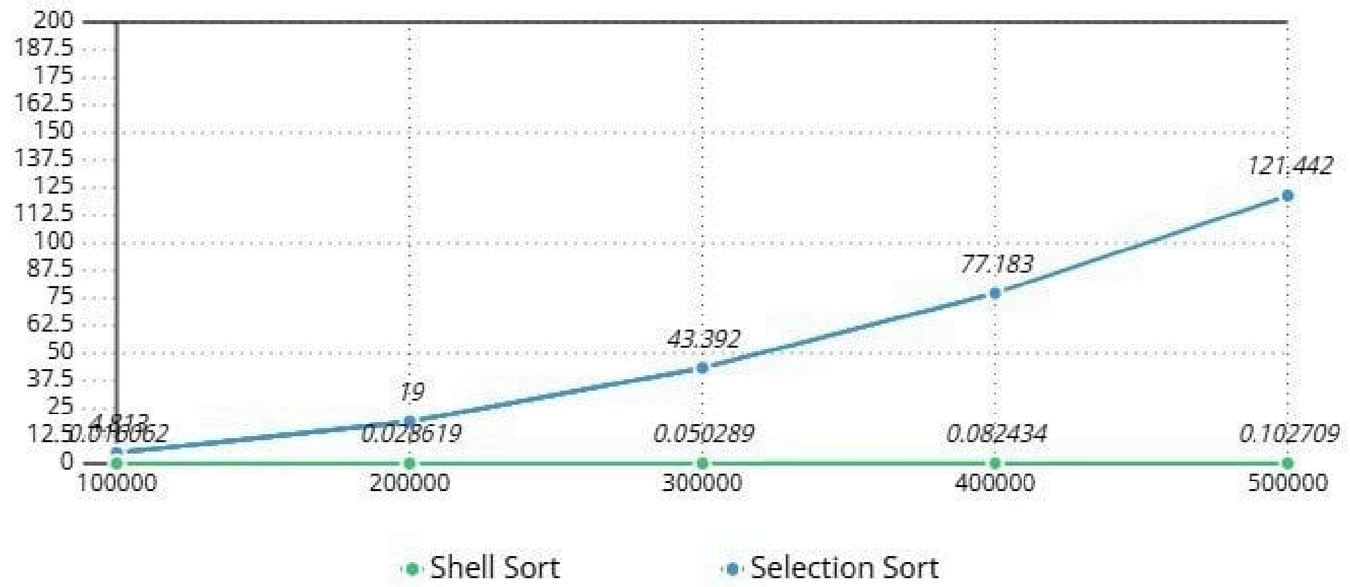
Memória: 500GB.

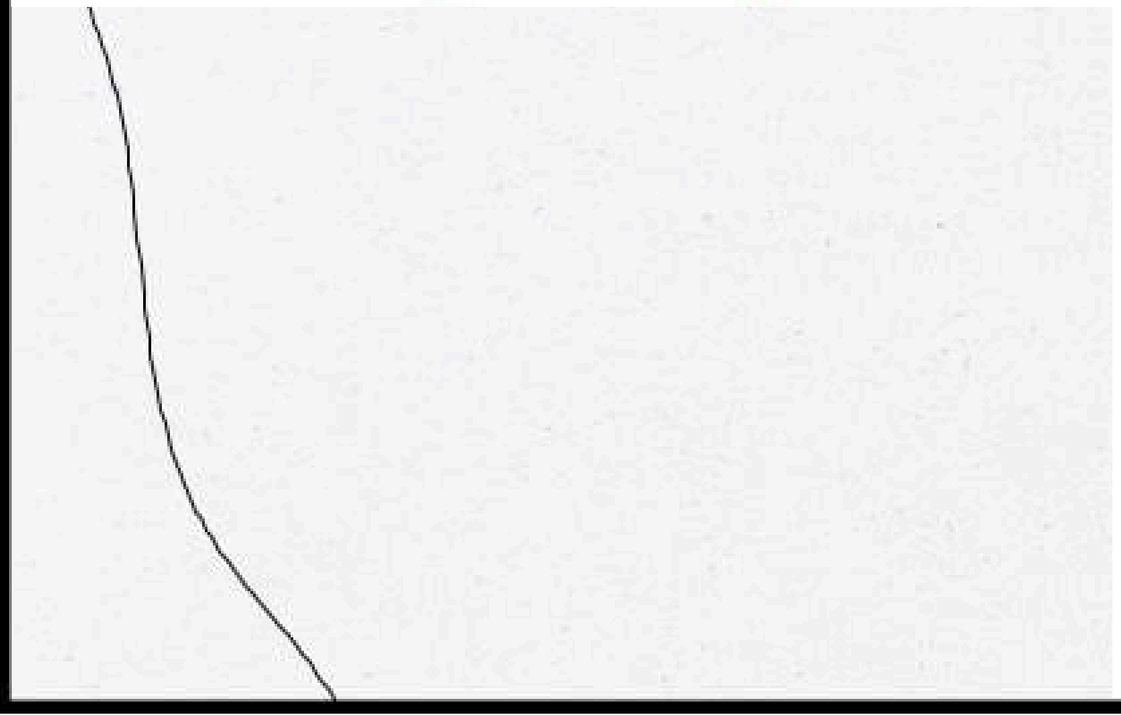
Resultados dos testes

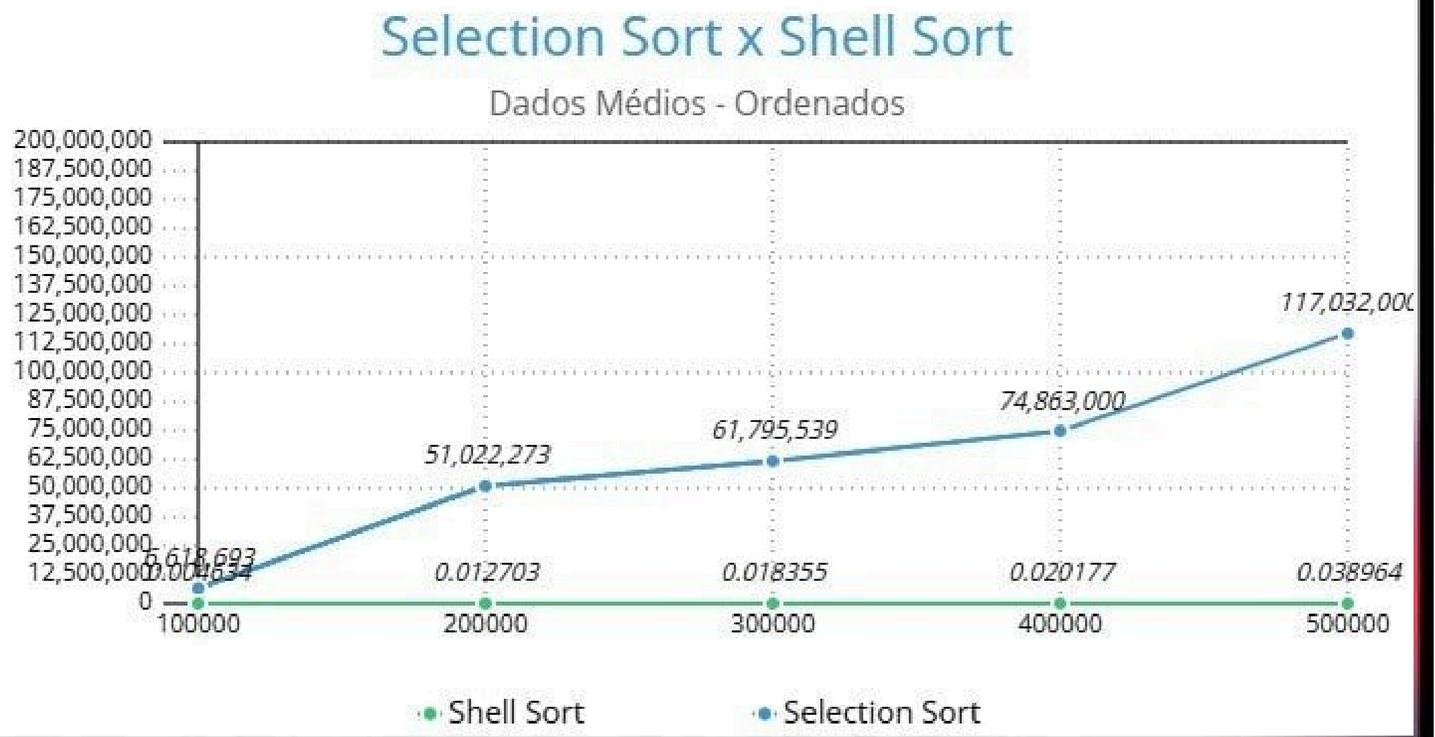




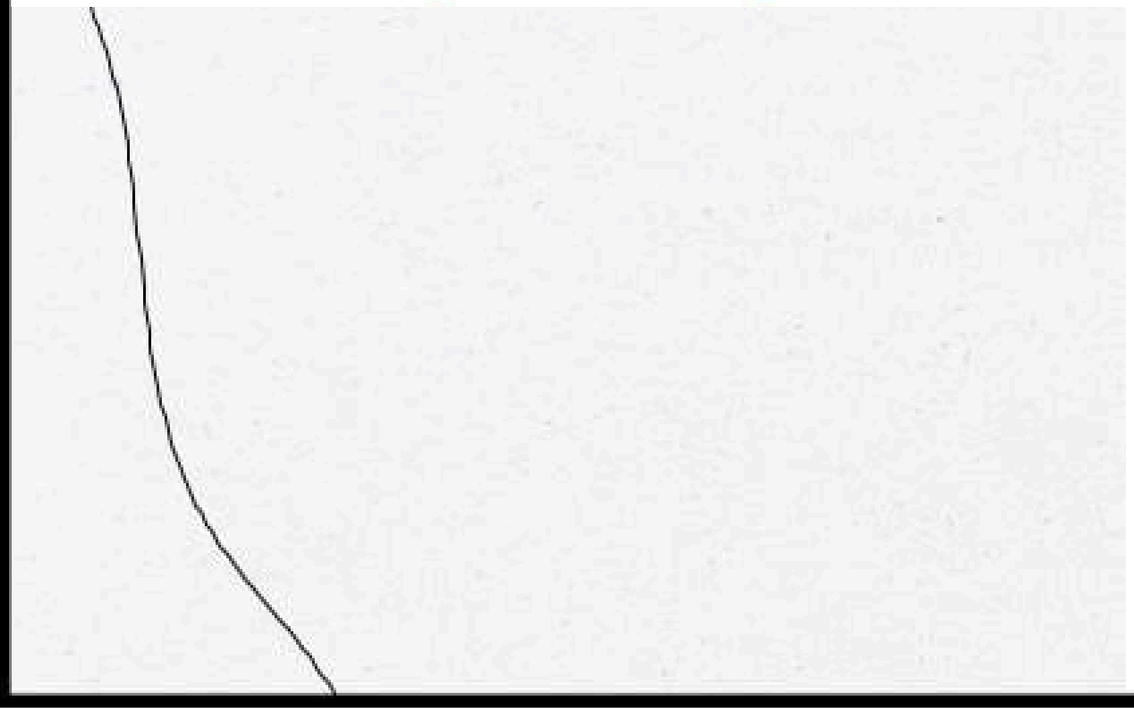
Resultados dos testes

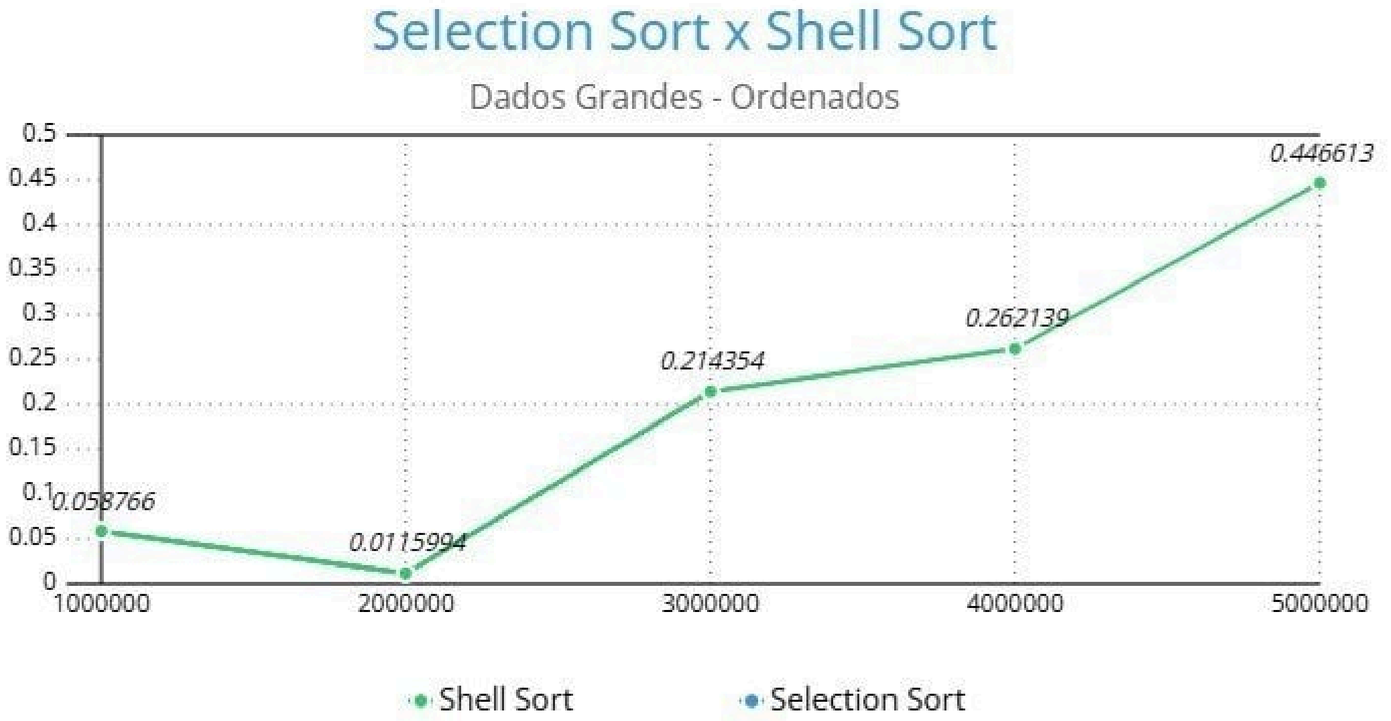






Resultados dos testes





Análise comparativa dos resultados

De acordo com o apresentado, compreende-se que o algoritmo Shell Sort, independentemente da quantidade de dados, possui uma eficiência grande, visto que o tempo de processamento em momento algum chegou a marca dos 2 segundos.

Em contrapartida, o algoritmo Selection Sort, já com dados medianos, facilmente chega a marca dos 2 minutos de processamento. Além disso, não foi possível obter os resultados desse algoritmo para dado grandes, visto que o algoritmo trava o compilador, o que impossibilita a obtenção de resultados.

Conclusão

- O Selection Sort é um algoritmo simples e fácil de implementar. No entanto, ele é ineficiente para grandes conjuntos de dados devido à sua complexidade de tempo O(n2). Cada iteração encontra o menor elemento no subarray não ordenado e o coloca na posição correta, resultando em muitas comparações e trocas desnecessárias.
- Vantagens: Simplicidade na implementação e entendimento, consistente e previsível em termos de desempenho, não requer memória adicional significativa.
- Desvantagens: Ineficiente para listas grandes devido à sua complexidade quadrática e não é um algoritmo estável.

- O Shell Sort é uma melhoria significativa em relação ao Insertion Sort, especialmente para listas maiores. Utiliza uma estratégia de inserção por intervalos (gaps) decrescentes, o que reduz consideravelmente o número de comparações e movimentações. Embora a complexidade exata dependa da escolha dos gaps, na prática, o Shell Sort oferece um desempenho muito melhor do que o Selection Sort.
- Vantagens: Mais eficiente que o Selection Sort e adequado para listas de tamanho médio, simples de implementar e adaptar a diferentes tipos de dados, reduz o número de comparações e movimentações necessárias.
- Desvantagens: O desempenho pode variar dependendo da escolha da sequência de gaps e não é um algoritmo estável.

Conclusão

Recomendações de Uso

- Quando Usar Selection Sort:
- Educação: Ideal para fins educacionais devido à sua simplicidade e fácil entendimento, permitindo aprender conceitos básicos de algoritmos de ordenação.
- Pequenos Conjuntos de Dados: Adequado para pequenas listas onde a simplicidade e previsibilidade são mais importantes que a eficiência.
- Memória Limitada: Útil em sistemas onde a memória adicional é restrita, pois funciona in-place.

- Quando Usar Shell Sort:
- Tamanhos Médios de Dados: Adequado para listas de tamanho médio onde é necessário um bom compromisso entre simplicidade e eficiência.
- Desempenho Melhorado: Quando se precisa de uma ordenação mais rápida do que o oferecido pelo Selection Sort e Insertion Sort.
- Simplicidade Relativa: Em situações onde algoritmos mais complexos como Quick Sort ou Merge Sort são desnecessários ou complicados demais.

- A escolha entre Selection Sort e Shell Sort depende do contexto de uso. O Selection Sort é mais simples e útil para conjuntos de dados pequenos ou fins educacionais, enquanto o Shell Sort oferece melhor desempenho prático para listas de tamanho médio. Em aplicações onde a eficiência é crucial, e a simplicidade do código é ainda um fator, o Shell Sort é geralmente preferido.