Ministério da Educação
Universidade Tecnológica Federal Do Paraná
Campus Campo Mourão

LUIZA BATISTA BASSETO

ALGORITMOS E ESTRUTURA DE DADOS 1

Avaliação 2 - Algoritmos de Ordenação e Busca

CAMPO MOURÃO

2025

Sumário

Métodos utilizados

O código foi implementado em C++, exceto pelos gráficos, que foram gerados pela biblioteca matplotlib em Python, e dividido em duas fases:

1. Testes iniciais com vetores pequenos, definidos manualmente, para verificar a corretude dos algoritmos.
2. Testes reais, com vetores gerados aleatoriamente e salvos em arquivos binários de diferentes tamanhos.

Os algoritmos de ordenação utilizados foram:

- Bubble Sort (puro e otimizado)
- Selection Sort (puro e otimizado)
- Insertion Sort

E as buscas utilizadas foram:

- Busca Linear
- Busca Binária

Os vetores foram preenchidos com números aleatórios e os tempos de execução foram medidos com precisão utilizando a biblioteca <chrono>.

O código completo com os algoritmos, geração de vetores, leitura de arquivos binários e medições de tempo está documentado e organizado em arquivos .cpp, disponíveis no repositório GitHub do projeto.

Os tamanhos escolhidos para cada arquivo, baseado nas entradas (pequena, média e grande) foram n1 (pequena) = 12.000 números aleatórios gerados; n2 (média) = 70.000 números aleatórios e, por último, n3 (grande) = 100.000 números gerados. Estes tamanhos foram escolhidos após vários testes com os algoritmos, e feito uma aproximação, baseado na média de tempos dos algoritmos, para chegar nos tempos certos para cada um dos tamanhos, ou seja, todos os resultados terão

tempos de execução próximos aos resultados requisitados de 1s (pequeno), 30s (médio) e 3min (grandes), podendo ter valores tanto acima dela quanto abaixo.

Análise Crítica e Gráficos

O trabalho teve como objetivo analisar o desempenho de diferentes algoritmos de ordenação e busca, tanto em vetores pequenos (para validação lógica) quanto em grandes volumes de dados (para análise de eficiência).

A primeira main.cpp foi utilizada para testes com vetores pequenos, toda ela foi gerada pelo Gemini, com o objetivo de verificar a correção dos algoritmos, isto é, se ao final das funções de ordenação os vetores estavam devidamente ordenados, e se as buscas retornavam os resultados esperados.

A segunda main.cpp, contida no arquivo-fonte final do trabalho, implementa uma abordagem mais robusta: os vetores são preenchidos com números aleatórios, salvos e lidos em arquivos binários, com tamanhos variados. Isso permitiu a medição precisa de desempenho em termos de tempo de execução, número de comparações e trocas, possibilitando uma análise prática e realista.

Análise baseado no arquivo de vetores pequenos

Link do arquivo mainGemini (GitHub): O arquivo está completamente comentado, devido se tratar de outra main, caso fosse deixado completamente descomentado, haveria conflitos de mains para a execução do programa.

Saída do Terminal de acordo com esse main.cpp com vetores prontos

```
===== Testando Algoritmos de Ordenação com Vetores Pequenos =====

--- Caso de Teste 1 ---
Vetor Original: { 5, 1, 4, 2, 8 }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: { 1, 2, 4, 5, 8 }
  Trocas: 4, Comparacoes: 9
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: { 1, 2, 4, 5, 8 }
  Trocas: 4, Comparacoes: 10
  Status: Ordenacao CORRETA.

  Aplicando InsertionSort...
  Vetor Ordenado: { 1, 2, 4, 5, 8 }
  Trocas: 4, Comparacoes: 7
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: { 1, 2, 4, 5, 8 }
  Trocas: 2, Comparacoes: 7
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: { 1, 2, 4, 5, 8 }
  Trocas: 2, Comparacoes: 10
  Status: Ordenacao CORRETA.
```

```
-------------------------------------

--- Caso de Teste 2 ---
Vetor Original: { 3, 1, 2 }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: { 1, 2, 3 }
  Trocas: 2, Comparacoes: 3
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: { 1, 2, 3 }
  Trocas: 2, Comparacoes: 3
  Status: Ordenacao CORRETA.

  Aplicando InsertionSort...
  Vetor Ordenado: { 1, 2, 3 }
  Trocas: 2, Comparacoes: 3
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: { 1, 2, 3 }
  Trocas: 2, Comparacoes: 3
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: { 1, 2, 3 }
  Trocas: 2, Comparacoes: 3
  Status: Ordenacao CORRETA.
-------------------------------------

--- Caso de Teste 3 ---
Vetor Original: { 1, 2, 3, 4, 5 }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 0, Comparacoes: 4
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 0, Comparacoes: 10
  Status: Ordenacao CORRETA.

  Aplicando InsertionSort...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 0, Comparacoes: 4
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 0, Comparacoes: 10
  Status: Ordenacao CORRETA.
-------------------------------------

--- Caso de Teste 4 ---
Vetor Original: { 5, 4, 3, 2, 1 }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 10, Comparacoes: 10
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 10, Comparacoes: 10
  Status: Ordenacao CORRETA.

  Aplicando InsertionSort...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 10, Comparacoes: 10
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 2, Comparacoes: 7
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: { 1, 2, 3, 4, 5 }
  Trocas: 2, Comparacoes: 10
  Status: Ordenacao CORRETA.
-------------------------------------

--- Caso de Teste 5 ---
Vetor Original: {  }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: {  }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: {  }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.
```

```
  Aplicando InsertionSort...
  Vetor Ordenado: {  }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: {  }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: {  }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.
-------------------------------------

--- Caso de Teste 6 ---
Vetor Original: { 7 }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: { 7 }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: { 7 }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando InsertionSort...
  Vetor Ordenado: { 7 }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: { 7 }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: { 7 }
  Trocas: 0, Comparacoes: 0
  Status: Ordenacao CORRETA.
-------------------------------------

--- Caso de Teste 7 ---
Vetor Original: { 2, 3, 2, 1, 3, 1 }

  Aplicando BubbleSortOptimized...
  Vetor Ordenado: { 1, 1, 2, 2, 3, 3 }
  Trocas: 8, Comparacoes: 15
  Status: Ordenacao CORRETA.

  Aplicando BubbleSort...
  Vetor Ordenado: { 1, 1, 2, 2, 3, 3 }
  Trocas: 8, Comparacoes: 15
  Status: Ordenacao CORRETA.

  Aplicando InsertionSort...
  Vetor Ordenado: { 1, 1, 2, 2, 3, 3 }
  Trocas: 8, Comparacoes: 12
  Status: Ordenacao CORRETA.

  Aplicando SelectionSortOptimized...
  Vetor Ordenado: { 1, 1, 2, 2, 3, 3 }
  Trocas: 2, Comparacoes: 9
  Status: Ordenacao CORRETA.

  Aplicando SelectionSort...
  Vetor Ordenado: { 1, 1, 2, 2, 3, 3 }
  Trocas: 2, Comparacoes: 15
  Status: Ordenacao CORRETA.
-------------------------------------


===== Testando Algoritmos de Busca com Vetores Pequenos =====

Vetor para Busca (Nao Ordenado): { 50, 10, 40, 20, 80, 70, 30, 60 }
Vetor para Busca (Ordenado): { 10, 20, 30, 40, 50, 60, 70, 80 }

--- Testando Busca Linear ---
    LinearSearch: Elemento 20 encontrado no indice 3. (Comparacoes: 4)
       LinearSearch: Elemento 77 NAO encontrado. (Comparacoes: 8)
    LinearSearch: Elemento 50 encontrado no indice 0. (Comparacoes: 1)
    LinearSearch: Elemento 10 encontrado no indice 1. (Comparacoes: 2)
    LinearSearch: Elemento 80 encontrado no indice 4. (Comparacoes: 5)
       LinearSearch: Elemento 99 NAO encontrado. (Comparacoes: 8)

--- Testando Busca Binaria (requer vetor ordenado) ---
  Testando em vetor CORRETAMENTE ordenado:
      BinarySearch: Elemento 20 encontrado no indice 1. (Comparacoes: 2)
```

```
    BinarySearch: Elemento 77 NAO encontrado. (Comparacoes: 4)
    BinarySearch: Elemento 50 encontrado no indice 4. (Comparacoes: 3)
    BinarySearch: Elemento 10 encontrado no indice 0. (Comparacoes: 3)
    BinarySearch: Elemento 80 encontrado no indice 7. (Comparacoes: 4)
    BinarySearch: Elemento 99 NAO encontrado. (Comparacoes: 4)

Testando em vetor NAO ordenado (comportamento pode ser inesperado):
    BinarySearch (em nao ordenado): Elemento 20 encontrado no indice 3 (pode ser sorte ou erro). (Comparacoes: 1)
-------------------------------------
```

Os testes iniciais com vetores pequenos confirmaram a correta ordenação dos dados por todos os algoritmos implementados (Bubble Sort puro e otimizado, Selection Sort puro e otimizado, e Insertion Sort).

O algoritmo Insertion Sort demonstrou o menor número de comparações nos casos em que o vetor estava quase ordenado (casos 1 e 3), refletindo sua eficiência em situações de dados parcialmente ordenados.

Já o Selection Sort Otimizado apresentou melhor desempenho no pior caso (vetor completamente invertido), fazendo menos trocas (7 contra 10 dos outros algoritmos) embora o número de comparações tenha permanecido alto. Isso destaca a eficiência da otimização para reduzir operações custosas.

O Bubble Sort Otimizado mostrou melhoria significativa em relação ao Bubble Sort puro, especialmente por conta da flag swapped, que interrompe o processo assim que o vetor está ordenado, reduzindo o número de comparações em vetores já ordenados (caso 3).

Esses resultados confirmam que otimizações focadas em interrupção antecipada e redução de trocas podem melhorar significativamente o desempenho em casos específicos, mas não alteram a complexidade geral dos algoritmos.

Análise baseada no arquivo oficial (Arquivos Binários)

Link do arquivo Main Oficial (GitHub): Essa foi a main feita baseada nos quesitos dos trabalho com a geração de arquivos de binários, em seguida a ordenação para cada um dos tamanhos e sendo executado cada um dos tipos de algoritmos e buscas, e por fim, a criação de novos arquivos binários ordenados.

Saída do Terminal para a main.cpp com arquivos binários

PARA O TAMANHO 1, de n1=12000, com demora de aproximadamente 1s

Tempo gasto com Bubble Sort Otimizado: 1.45114 segundos

Trocas: 35709112, Comparações: 71983704

Tempo gasto com Bubble Sort Puro: 1.33083 segundos

Trocas: 35709112, Comparações: 71994000

Tempo gasto com Selection Sort Otimizado: : 1.24734 segundos

Trocas: 11971, Comparações: 71993922

Tempo gasto com Insertion Sort: 0.435547 segundos

Trocas: 35709112, Comparações: 35721107

Busca Linear em vetor não ordenado

Resultado Index: 613 | Comparações: 614 | Tempo Gasto: 1031.000000 microsegundos

Busca Binária em vetor não ordenado

Resultado Index: -1 | Comparações: 13 | Tempo Gasto: 0.000000 microsegundos

Tempo gasto com Selection Sort: 0.702620 segundos

Trocas: 11971, Comparações: 71994000

Busca Linear em vetor ordenado

Resultado Index: 97 | Comparações: 98 | Tempo Gasto: 0.000000 microsegundos

Busca Binária em vetor ordenado

Resultado Index: 103 | Comparações: 10 | Tempo Gasto: 0.000000 microsegundos

---

PARA O TAMANHO 2, de n2=70000, com demora de aproximadamente 30s

Tempo gasto com Selection Sort: 18.218173 segundos

Trocas: 69911, Comparações: 2449965000

Tempo gasto com Selection Sort Otimizado: 39.969767 segundos

Trocas: 69911, Comparações: 2449962074

Busca Linear em vetor não ordenado

Resultado Index: 613 | Comparações: 614 | Tempo Gasto: 0.000000 microsegundos

Busca Binária em vetor não ordenado

Resultado Index: -1 | Comparações: 16 | Tempo Gasto: 0.000000 microsegundos

Tempo gasto com Insertion Sort: 15.741638 segundos

Trocas: 1223487348, Comparações: 1223557343

Busca Linear em vetor ordenado

Resultado Index: 567 | Comparações: 568 | Tempo Gasto: 0.000000 microsegundos

Busca Binária em vetor ordenado

Resultado Index: 613 | Comparações: 10 | Tempo Gasto: 0.000000 microsegundos

Tempo gasto com Bubble Sort Otimizado: 47.826009 segundos

Trocas: 1223487348, Comparações: 2449947980

Tempo gasto com Bubble Sort Puro: 111.310902 segundos

Trocas: 1223487348, Comparações: 2449965000

PARA O TAMANHO 3, de n3=100000, com demora de aproximadamente 3min

Tempo gasto com Selection Sort: 90.830032 segundos

Trocas: 99887, Comparações: 4999950000

Tempo gasto com Insertion Sort: 73.401721 segundos

Trocas: 2506572675, Comparações: 2506672670

Tempo gasto com Selection Sort Otimizado: 213.950280 segundos

Trocas: 99887, Comparações: 4999944435

Tempo gasto com Bubble Sort Otimizado: 232.451618 segundos

Trocas: 2506572675, Comparações: 4999880994

Tempo gasto com Bubble Sort Puro: 214.831378 segundos

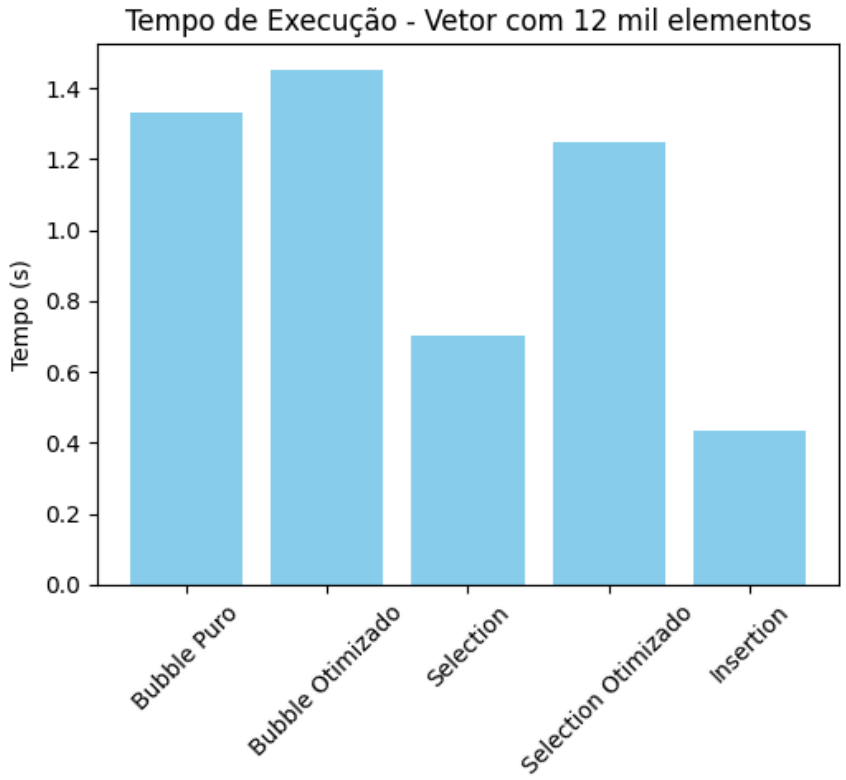
Trocas: 2506572675, Comparações: 49999500

Análise dos Algoritmos

Pode-se concluir após a observação da saída do terminal para os arquivos binários que os testes com vetores maiores, gerados e lidos dos arquivos, revelaram diferenças mais marcantes no desempenho dos algoritmos, incluindo o tempo de execução, número de trocas e comparações.

O Insertion Sort foi eficiente, sendo o melhor dentre todos, em todos os tempos (n1, n2, n3), e que, embora seu número de trocas quanto de comparações, foram bem mais elevados do que do Selection Sort, por exemplo, seu tempo não refletiu isso, se sobressaindo em todos os casos dentre os outros, como é apresentado na Figura 1 e também é possível perceber nas demais figuras.

Figura 1- Gráfico sobre o tempo de execução dos algoritmos com tamanho N1

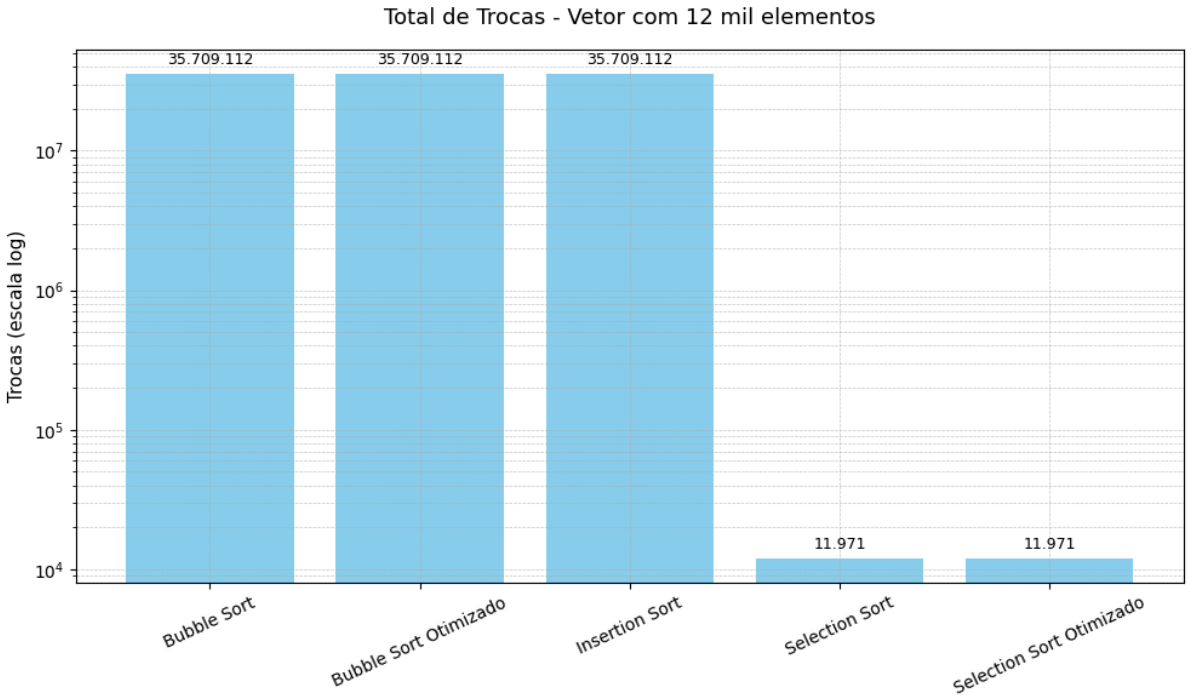


Fonte: Arquivo Próprio, 2025.

O algoritmo Selection Sort (puro) apresentou o segundo melhor tempo para esses vetores maiores, devido ao número relativamente baixo e constante de trocas, apesar de um elevado número de comparações. Essa característica o tornou mais eficiente que os outros algoritmos para esse volume de dados. Como é possível

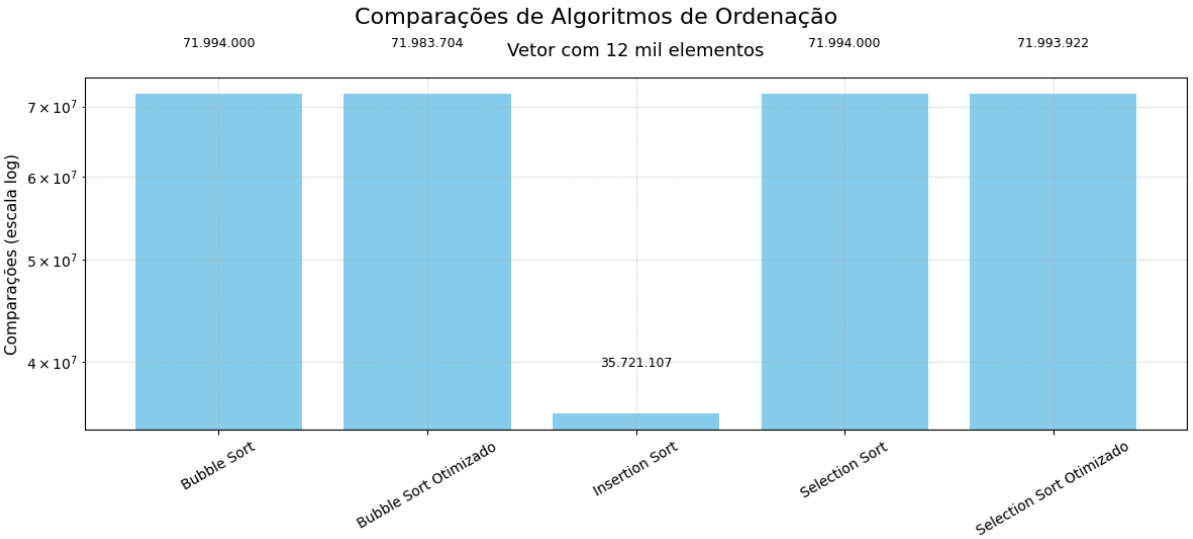
observar através da Figura 2 abaixo que mostra o número de comparações e a Figura 3 que mostra o número de trocas, ambos para o vetor N1, que possui 12.000 elementos.

Figura 2- Gráfico sobre o número de trocas de cada algoritmo para o vetor de tamanho N1



Fonte: Arquivo Próprio, 2025.

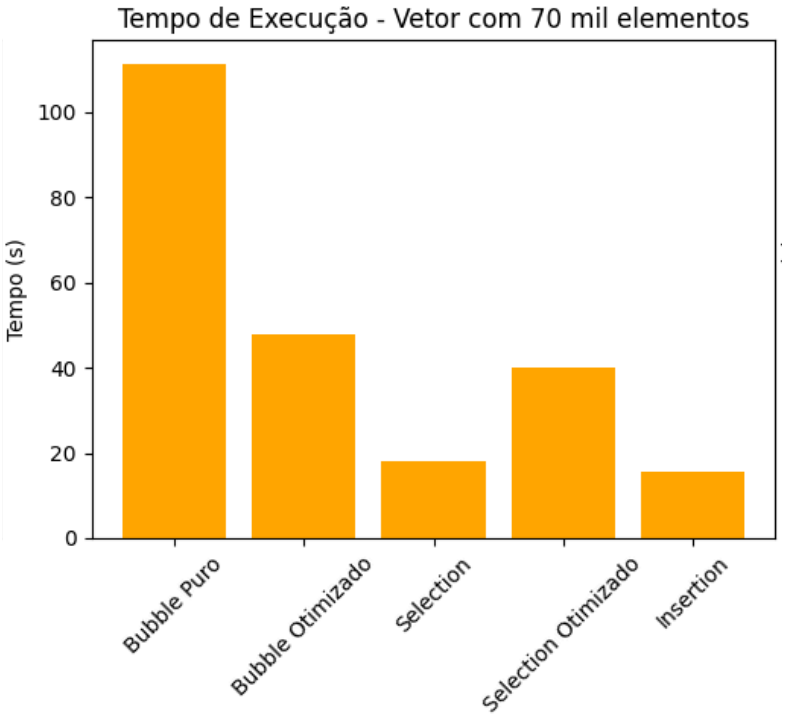
Figura 3- Gráfico sobre o número de comparações de cada algoritmo para o vetor de tamanho N1



Fonte: Arquivo Próprio, 2025.

Os algoritmos de bolha (Bubble Sort Otimizado e Puro), que comparam par a par, elementos adjacentes, apresentaram os piores tempos para vetores grandes (exemplo: 47s e 111s para 70.000 elementos, e ainda maior para 100.000), confirmando que esse algoritmo não é recomendado para grandes volumes de dados. Onde é possível observar por meio da Figura 4, seus altos tempos de execução quando comparados com os outros algoritmos como o Selection Sort e o Insertion Sort.

Figura 4 - Tempo de Execução dos Algoritmos com tamanho N2



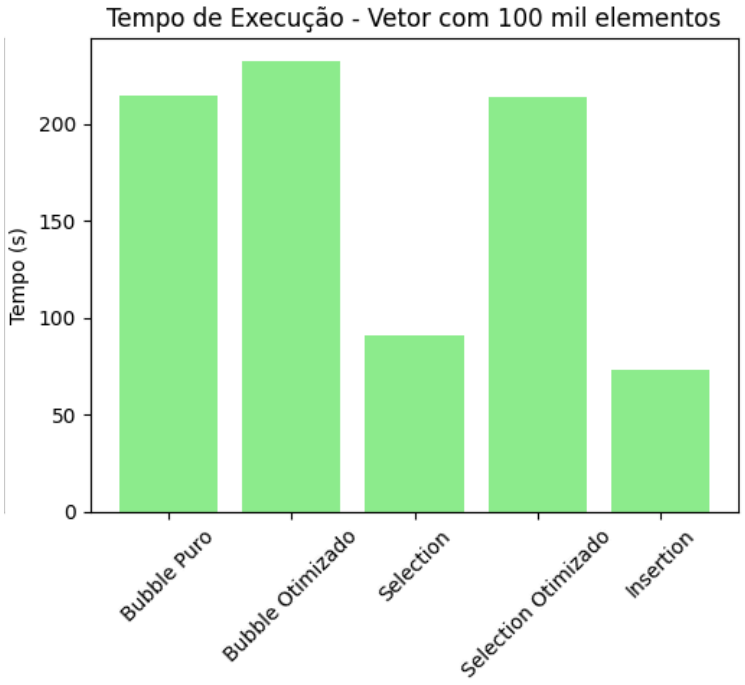
Fonte: Arquivo Próprio, 2025.

Uma “decepção” foi o algoritmo do Selection Sort Otimizado que não trouxe melhorias significativas no tempo, sendo até mais lento que o Selection Sort puro em grandes conjuntos, o que insinua que o custo da otimização pode superar seus benefícios para esse tamanho e tipo de dados, e mostra que para esses casos, o Selection Sort Otimizado, só funcionando adequadamente reduzindo o tempo para vetores já ordenados ou praticamente ordenados, caso contrário, o custo da otimização (parte do código a mais), acaba acarretando em uma complexidade

maior e assim aumentando seu tempo de execução, como é possível perceber em todos os gráficos das figuras vistas até este momento.

Por fim, para o maior vetor testado (100.000 elementos), os tempos foram elevados para todos os algoritmos, mas com o Insertion Sort e Selection Sort puro se destacando e sendo ainda mais rápidos que os otimizados e que o Bubble Sort, como é possível perceber por meio da Figura 5, que ilustra bem, como é desigual os tempos de execução do Selection e do Insertion Sort quando comparados com os algoritmos de bolha.

Figura 5 - Tempo de Execução dos Algoritmos com tamanho N3



Fonte: Arquivo Próprio, 2025.

Análise das Buscas

Sobre as buscas é possível perceber que a Busca Linear manteve desempenho estável, e foi eficaz tanto em vetores ordenados quanto não ordenados, embora com bem mais comparações e tempo de execução que a busca binária.

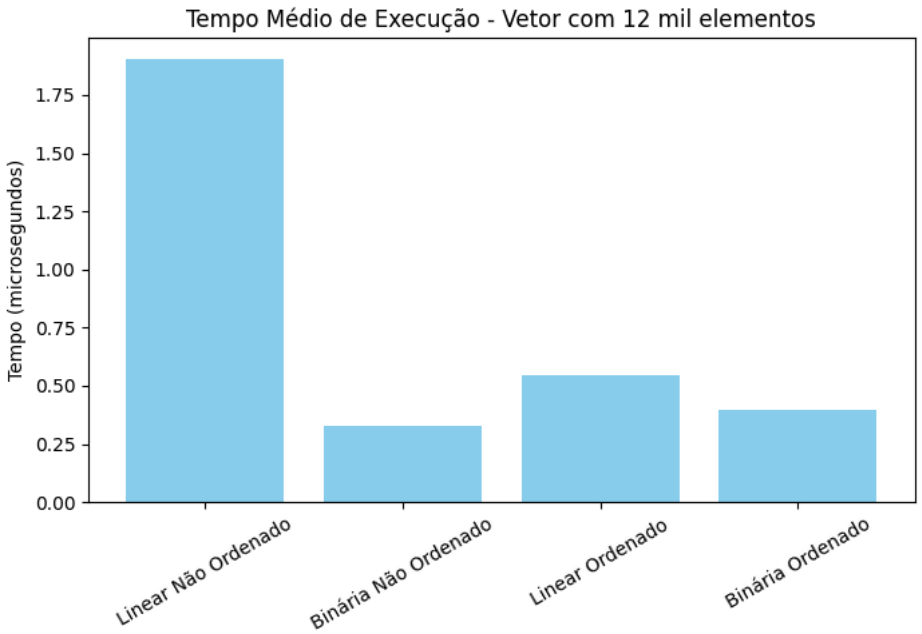
Já a busca binária funcionou corretamente apenas para vetores ordenados, como esperado, e apresentou número reduzido de comparações, refletindo sua complexidade logarítmica. A busca binária em vetor não ordenado retornou

resultados fora do index (-1), reforçando a necessidade de pré-ordenação para seu uso. Foram feitas buscas apenas para os vetores de tamanho N1 e N2, entretanto foram feitas buscas antes e depois das ordenações, para poder comparar os tempos.

Para a contagem correta do tempo de execução de ambas as buscas foi preciso ser feito um algoritmo a mais de repetição da busca n vezes para que o tempo de execução do algoritmo de busca fosse relevante o suficiente para o valor sair de 0. Então para resolver este problema foi utilizado o algoritmo 100.000 vezes dentro da função de contagem do tempo, podendo ser acessado pelo link da linha dessa execução (Link), e após a contagem, feito a média de acordo com essa quantidade de vezes executada, e estes valores foram apresentados nos gráficos abaixo.

Para o vetor de tamanho N1 (12.000):

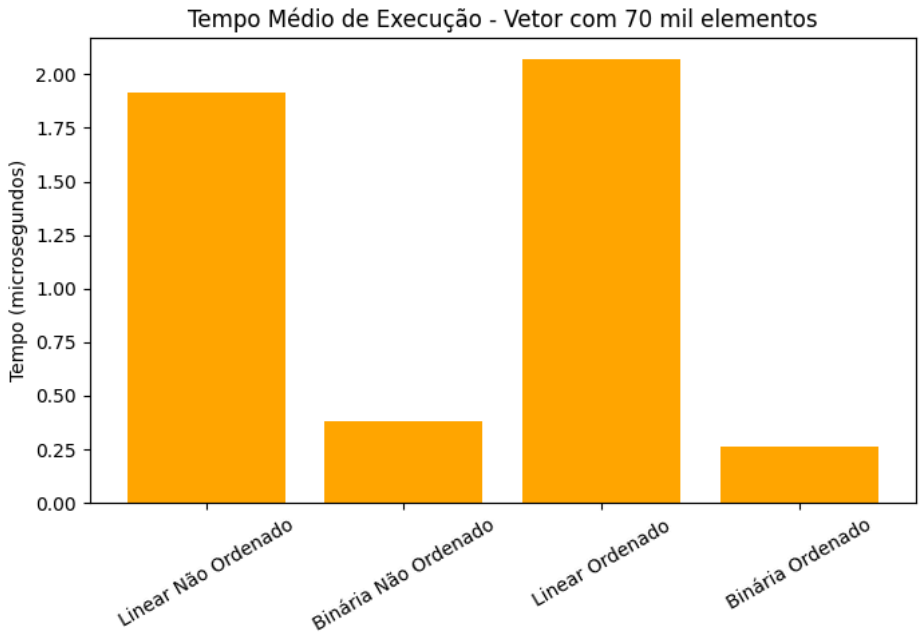
Tempo Médio de Execução das buscas com tamanho N1



Fonte: Arquivo Próprio, 2025.

Já para o vetor de tamanho N2 (70.000 elementos):

Tempo Médio de Execução das buscas com tamanho N2



Fonte: Arquivo Próprio, 2025.

Após a análise dos gráficos de buscas, é possível perceber que é enorme a discrepância entre os tempos de execução das buscas lineares e buscas binárias, a busca binária devido seu algoritmo muito mais complexo, executa a busca de forma muito mais eficiente que a busca linear. Entretanto, sua única dificuldade, é que o vetor precisa, obrigatoriamente, estar ordenado para funcionar corretamente, algo que pelo gráfico não é possível identificar.

Conclusão

Concluindo, após a análise das saídas dos terminais e dos gráficos dos algoritmos de ordenação e buscas, é possível afirmar que o algoritmo Insertion Sort apresentou o melhor desempenho geral em vetores grandes, mesmo com altos valores de comparações e trocas. No vetor de 200.000 elementos, foi mais eficiente que o Selection e Bubble, com tempo de execução consideravelmente menor.

Já o Selection Sort foi mais eficiente em vetores pequenos ou parcialmente ordenados. Em listas quase ordenadas, ele realiza poucas comparações e trocas, o que justifica seu bom desempenho. No vetor ordenado, no caso da main de vetores prontos e pequenos, sua versão otimizada apresentou menor número de trocas e comparações dentre todos os algoritmos.

O Bubble Sort, mesmo em sua versão otimizada, apresentou o pior desempenho em listas grandes e aleatórias. O algoritmo puro chegou a ultrapassar 200 segundos no maior teste. A otimização com a flag swapped mostrou ganhos apenas em vetores pequenos ou já ordenados, onde o algoritmo consegue encerrar precocemente.

Em relação às buscas, a Busca Linear se manteve estável em todos os casos, enquanto a Busca Binária só apresentou resultados corretos quando aplicada a vetores ordenados, como esperado, porém se destacando pelo número baixo de comparações, sendo mais eficiente que a linear.

Concluindo, algoritmos otimizados demonstram vantagem apenas em melhor caso (vetores ordenados), mas não mudam a complexidade geral do algoritmo. Portanto, a escolha do algoritmo deve considerar o tamanho e a ordenação prévia dos dados, pois isso impacta diretamente no desempenho prático.