JOSÉ DAVI VIANA FRANCELINO
THIAGO DE OLIVEIRA CORDEIRO

Análise empírica de algoritimos

Natal, RN
2020

JOSÉ DAVI VIANA FRANCELINO
THIAGO DE OLIVEIRA CORDEIRO

Análise empírica de algoritimos

Relatório técnico apresentado à disciplina de Estrutura de Dados Básicas I, como requisito parcial para obtenção de nota referente à unidade I.

Natal, RN
2020

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Um algoritmo é uma sequência finita de passos bem definidos e não ambíguos que para cada entrada de dados gera uma saída, onde busca-se solucionar algum problema. Assim, em conformidade com Cormen, Leiserson e Stein (2002), analisar um algoritmo significa prever os recursos que ele necessitará.

A previsão dos recursos necessários para os algoritmos de ordenação deste trabalho foi obtida a partir da análise empírica do tempo de execução de cada um dos algoritmos implementados. Desse modo, espera-se avaliar o comportamento assintótico dos algoritmos em relação ao seu tempo de execução. Ademais, cabe destacar que esse método de análise auxiliará na escolha do algoritmo ideal para a realização da tarefa de ordenar um arranjo proposto na descrição do trabalho, haja vista que permitirá apontar a solução com maior eficiência.

Outrossim, é relevante destacar que os algoritmos de ordenação são de suma importância para a resolução de problemas computacionais. Em síntese, eles servem para ordenar ou organizar uma lista/conjunto de números, palavras, caracteres ou outra estrutura de dados de acordo com algum critério estabelecido.

Para a realização da análise deste relatório, foram implementados sete algoritmos: o *Insertion Sort*, *Selection Sort*, *Bubble Sort*, *Shell Sort*, *Quick Sort*, *Merge Sort* e *Radix Sort* (LSD). Ainda mais, as análises de tempo foram executadas em um único computador, em apenas um sistema operacional, o GNU/Linux. Além disso, com o fito de facilitar a comparação, os dados obtidos foram inseridos em gráficos.

Nos tópicos subsequentes, é mostrado com mais detalhes o método adotado neste trabalho, expondo quais cenários foram contemplados na análise, quais quantitativos de dados foram utilizados, ferramentas e afins. Por último, serão apresentados os resultados e a conclusão final da análise do trabalho.

2 MÉTODO

2.1 Materiais utilizados

Computador	Especificações
Lenovo ideapad S145-15IWL	Processador Intel Core i7-8565U CPU @ 1.80GHz Memória: 2 X 4GB de RAM 2400MHz

Tabela 1: Principais especificações técnicas do computador utilizado.

Sistema	Especificações
GNU/Linux	Debian 5.10.40-1 (x86_64) Kernel 5.10.0-7-amd64

Tabela 2: Detalhes técnicos dos sistemas operacionais utilizados.

2.2 Ferramentas de programação

Todos os 7 algoritmos foram implementados na linguagem de programação C++, no padrão ISO/IEC 14882:2011, ou apenas C++11.

A compilação dos algoritmos feitos neste trabalho foi realizada pelo compilador G++, dentro do ambiente GNU/Linux. Ademais, a versão do compilador foi a 10.2.1.

2.3 Algoritmos

Algorithm 1 Bubble sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A

```
1: while n > 1 do
2:     novo_n ← 0
3:     for i ← [1, n) do
4:         if A_i < A_{i-1} then
5:             swap(A_i, A_{i-1})
6:             novo_n ← i
7:         end if
8:     end for
9:     n ← novo_n
10: end while
```

Algorithm 2 Insertion Sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A.

```
1: for i ← [1, n) do
2:     auxiliar ← A_i
3:     j ← i
4:     while j > 0 and A_{j-1} > A_j do
5:         A_j = A_{j-1}
6:         j ← j − 1
7:     end while
8:     A_j = auxiliar
9: end for
```

Algorithm 3 Selection Sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A.

```
1: for i ← [0, n) do
2:     menor ← i
3:     for j ← [i + 1, n) do
4:         if A_menor > A_j then
5:             menor ← j
6:         end if
7:     end for
8:     swap(A_menor, A_i)
9: end for
```

Algorithm 4 Shell sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A

```
1: k ← 1
2: repeat
3:     gap ← 2 · ⌊ n / 2^(k+1) ⌋
4:     for i ← [gap, n) do
5:         aux ← A_i
6:         j ← i
7:         while j ≥ gap and aux < A_(j−gap) do
8:             A_j ← A_(j−gap)
9:             j ← j − gap
10:        end while
11:        A_j ← aux
12:    end for
13:    k ← k + 1
14: until gap > 1
```

Merge (auxiliar do merge sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$, um arranjo $L := \{L_0, L_1, \ldots, L_{l-1}\}$ e um arranjo $R := \{R_0, R_1, \ldots, R_{r-1}\}$

Result: Ordena o arranjo A

1: $i \leftarrow 0$
2: $j \leftarrow 0$
3: **while** $i < l$ **and** $j < r$ **do**
4: **if** $L_i < R_j$ **then**
5: $A_{i+j} \leftarrow L_i$
6: $i \leftarrow i + 1$
7: **else**
8: $A_{i+j} \leftarrow R_j$
9: $j \leftarrow j + 1$
10: **end if**
11: **end while**
12: **if** $i < l$ **then**
13: $\{A_{i+j}, A_{i+j+1}, \ldots, A_{n-1}\} \leftarrow \{L_i, L_{i+1}, \ldots, L_{l-1}\}$
14: **else if** $j < r$ **then**
15: $\{A_{i+j}, A_{i+j+1}, \ldots, A_{n-1}\} \leftarrow \{R_j, R_{j+1}, \ldots, R_{r-1}\}$
16: **end if**

Algorithm 5 Merge sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A

1: **if** $n \leq 1$ **then**
2: **return**
3: **end if**
4: $mid \leftarrow \dfrac{n}{2}$
5: $L \leftarrow \{A_0, A_1, \ldots, A_{mid-1}$
6: $R \leftarrow \{A_{mid}, A_{mid+1}, \ldots, A_{n-1}$
7: merge sort em L
8: merge sort em R
9: $merge(A, L, R)$

Partition (auxiliar do quick sort)

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$ e um valor *pivot*

Result: Divide A em duas partes: os que são menor que *pivot* e os que são maior que *pivot*

Output: A nova posição de p

1: $slow \leftarrow 0$
2: $fast \leftarrow 0$
3: **while** $fast < n$ **do**
4: **if** $A_{fast} < pivot$ **then**
5: $swap(A_{fast}, A_{slow})$
6: $slow \leftarrow slow + 1$
7: **end if**
8: $fast \leftarrow fast + 1$
9: **end while**
10: $swap(A_{slow}, pivot)$
11: **return** *slow*

Algorithm 6 Quick sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A

1: **if** $n \leq 1$ **then**
2: **return**
3: **end if**
4: $mid \leftarrow \dfrac{n}{2}$
5: **if** $A_{n-1} < A_0$ **then**
6: $swap(A_{n-1}, A_0)$
7: **end if**
8: **if** A_{mid}, A_0 **then**
9: $swap(A_{mid}, A_0)$
10: **end if**
11: **if** A_{mid}, A_{n-1} **then**
12: $swap(A_{mid}, A_{n-1})$
13: **end if**
14: $pivot \leftarrow partition(A, A_{n-1})$
15: Quick sort no arranjo $\{A_0, A_1, \ldots, A_{pivot-1}\}$
16: Quick sort no arranjo $\{A_{pivot+1}, A_{pivot+2}, \ldots, A_{n-1}\}$

Algorithm 7 Radix Sort

Input: Um arranjo $A := \{A_0, A_1, \ldots, A_{n-1}\}$

Result: Ordena o arranjo A.

1: $maior \leftarrow max(A_0, A_{n-1})$
2: $divisor \leftarrow 1$
3: **while** $maior > 0$ **do**
4: vetor de vetores: $buckets := \{B_0, B_1, \ldots, B_9\}$
5: $contador \leftarrow 0$
6: **for** $j \leftarrow [0, n)$ **do**
7: $digito \leftarrow \frac{A_j}{divider}\%10$
8: adicionar o elemento A_j ao arranjo *buckets* na posição *digito*.
9: **end for**
10: **for** $i \leftarrow [0, 9]$ **do**
11: $quantidade \leftarrow$ tamanho de $buckets[i]$
12: **for** $w \leftarrow 0, quantidade$ **do**
13: $A_{contador} \leftarrow$ buckets$_{iw}$
14: $contador \leftarrow contador + 1$
15: **end for**
16: **end for**
17: $maior \leftarrow \frac{maior}{10}$
18: $divisor \leftarrow divisor * 10$
19: **end while**

2.4 Método de comparação

Para comparar os algoritmos, executamos os sete em seis cenários diferentes: com o arranjo em ordem não decrescente, com o arranjo em ordem não crescente, com o arranjo em ordem aleatória, com o arranjo 75% ordenado, com o arranjo 50% ordenado, com o arranjo 25% ordenado.

2.4.1 Gerando os dados

2.4.1.1 Em ordem não decrescente

A fim de gerar dados mais precisos, criamos o vetor em ordem decrescente de forma aleatória. Para fazer isso, utilizamos o seguinte algoritimo:

Algorithm 8 Geração de dados em ordem não decrescente

1: $A \leftarrow \{A_0, A_1, \ldots, A_{n-1}\}$
2: $A_0 \leftarrow random\left(0, \frac{INT\_MAX}{n}\right)$
3: **for** $i \leftarrow [1, n)$ **do**
4: $\quad A_i \leftarrow A_{i-1} + random\left(0, \frac{INT\_MAX}{n}\right)$
5: **end for**

Com esse algoritmo, podemos garantir que o arranjo estará em ordem não decrescente e que não acontecerá um *overflow* de inteiro, já que, no pior dos casos, o último valor será INT_MAX. Entretanto, ele não dá a mesma probabilidade para cada inteiro ser sorteado, os valores longe das extremidades terão chances maiores de estarem nesse arranjo.

2.4.1.2 Em ordem não crescente

Da mesma forma do cenário anterior, os dados desse cenário foram gerados aleatoriamente.

Algorithm 9 Geração de dados em ordem não crescente

1: $A \leftarrow \{A_0, A_1, \ldots, A_{n-1}\}$
2: $A_0 \leftarrow INT\_MAX - random\left(0, \frac{INT\_MAX}{n}\right)$
3: **for** $i \leftarrow [1, n)$ **do**
4: $\quad A_i \leftarrow A_{i-1} - random\left(0, \frac{INT\_MAX}{n}\right)$
5: **end for**

Com esse algoritmo, podemos garantir que o arranjo estará em ordem não crescente e que todos os valores serão positivos, já que, no pior dos casos, o último valor será 0. Entretanto, da mesma forma que no arranjo não decrescente, valores longe das extremidades terão chances maiores de estarem nesse arranjo.

2.4.1.3 Em ordem aleatória

Algorithm 10 Geração de dados em ordem aleatória

1: $A \leftarrow \{A_0, A_1, \ldots, A_{n-1}\}$
2: **for** $i \leftarrow [0, n)$ **do**
3: $\quad A_i \leftarrow random\,(0, INT\_MAX)$
4: **end for**

2.4.1.4 Com uma porcentagem dos valores ordenados

Para esse tipo de cenário, geramos primeiramente um arranjo A em ordem não decrescente com o mesmo algoritmo utilizado anteriormente e aplicamos o seguinte algoritmo, considerando *percentage* como a porcentagem dos elementos do arranjo que devem ficar fora de ordem:

Algorithm 11 Geração de dados com uma porcentagem dos valores ordenados

1: $I \leftarrow \{0, 1, \ldots, n-1\}$
2: embaralha I
3: $i \leftarrow 0$
4: **while** $i < percentage * n$ **do**
5: $\quad swap(A_{I_i}, A_{I_{i+1}})$
6: $\quad i \leftarrow i + 2$
7: **end while**

Com esse algoritmo, podemos garantir que $1 - percentage$ dos elementos do arranjo estarão na sua posição final após o vetor ser ordenado, já que embaralhamos *percentage* deles.

2.4.2 Temporizando os algoritmos

Em cada cenário, os algoritmos foram executados com 25 tamanhos de arranjos diferentes, variando de 1.000 até 50.000 uniformemente. Para evitar ruído no gráfico, para cada arranjo gerado com os métodos discutidos na seção anterior, cada um dos sete algoritmos foram executados 5 vezes e a média dos tempos de execução foi armazenada.

3 RESULTADOS

Após rodar os testes para todos os cenários propostos e para cada algoritmo, os tempos de execução foram armazenados e foram gerados gráficos com o objetivo de comparar os dados obtidos. Para cada cenário, foram gerados dois gráficos: um comparando os algoritmos de complexidade $O(n^2)$ e outro comparando os algoritmos de complexidade $O(n \log n)$. O *radix sort*, que é o único com a complexidade diferente, foi incluído junto com os algoritmos de complexidade $O(n \log n)$. Além disso, alguns gráficos estão com escala logarítmica no eixo y e outros com escala linear, dependendo de que forma ele seria melhor visualizado.

3.1 Arranjo em ordem não decrescente

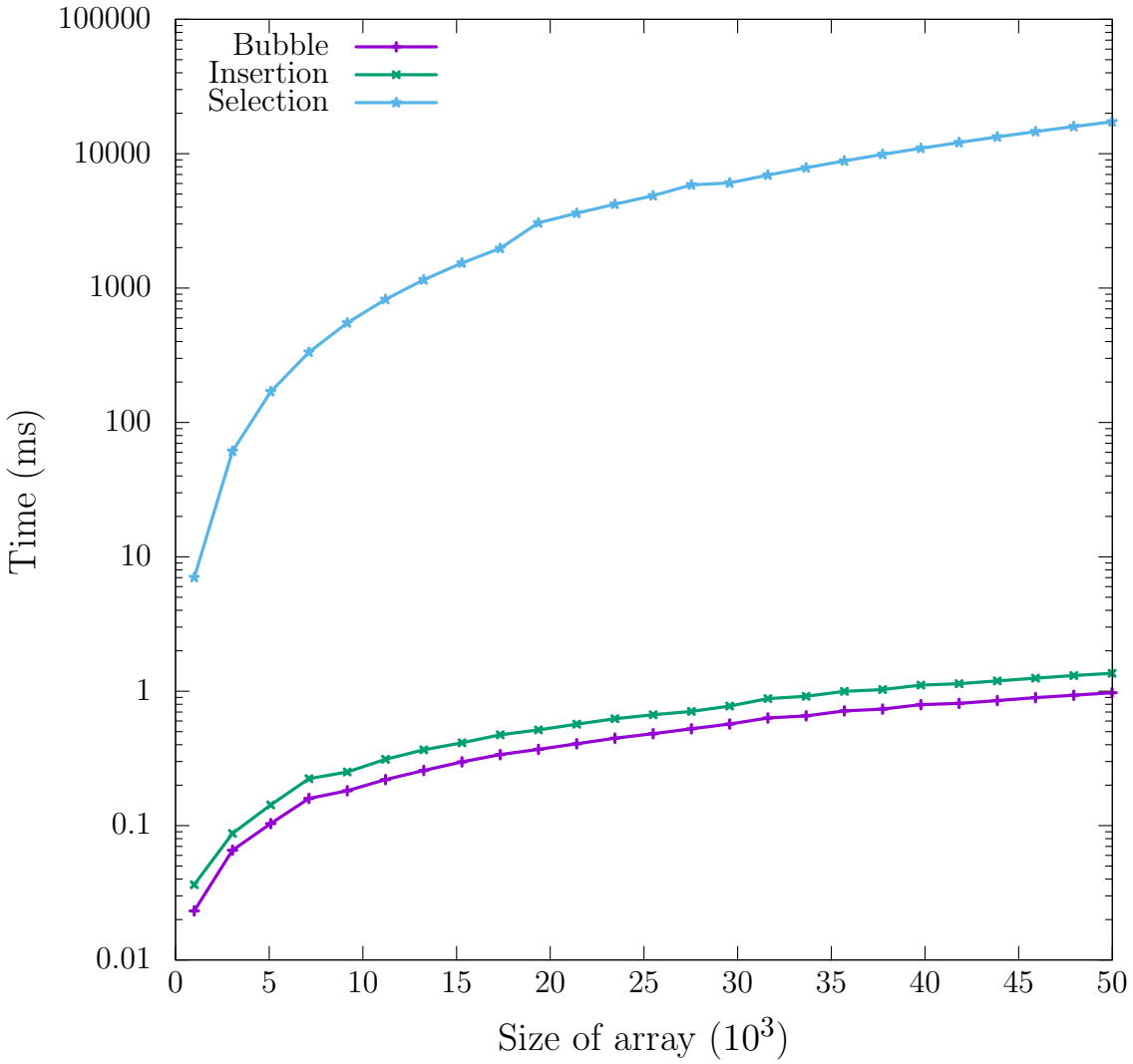

Figura 1: Gráfico dos algoritmos $O(n^2)$ para arranjo não decrescente

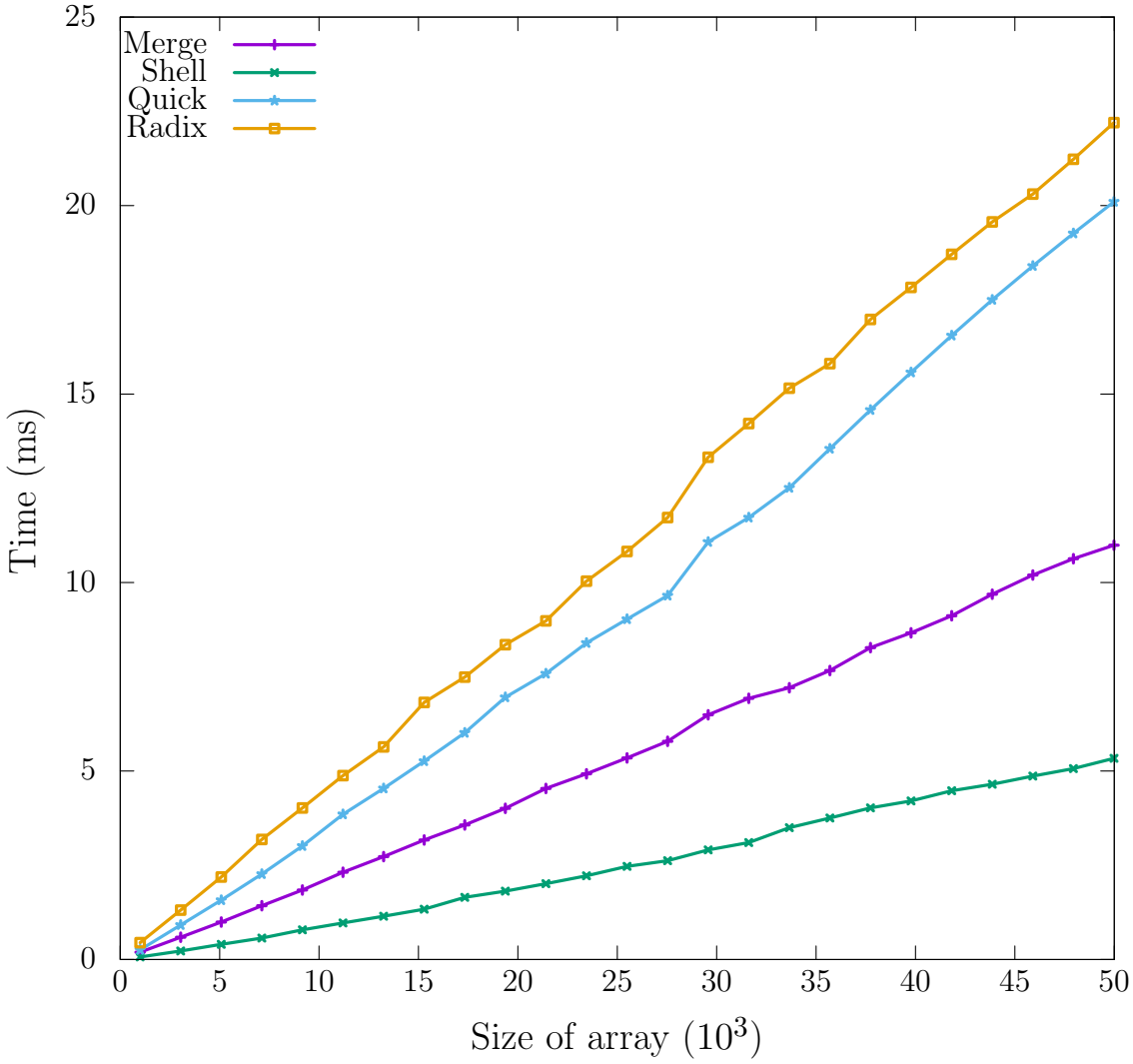

Figura 2: Gráfico dos algoritmos $O(n \log n)$ para arranjo não decrescente

3.2 Arranjo em ordem não crescente

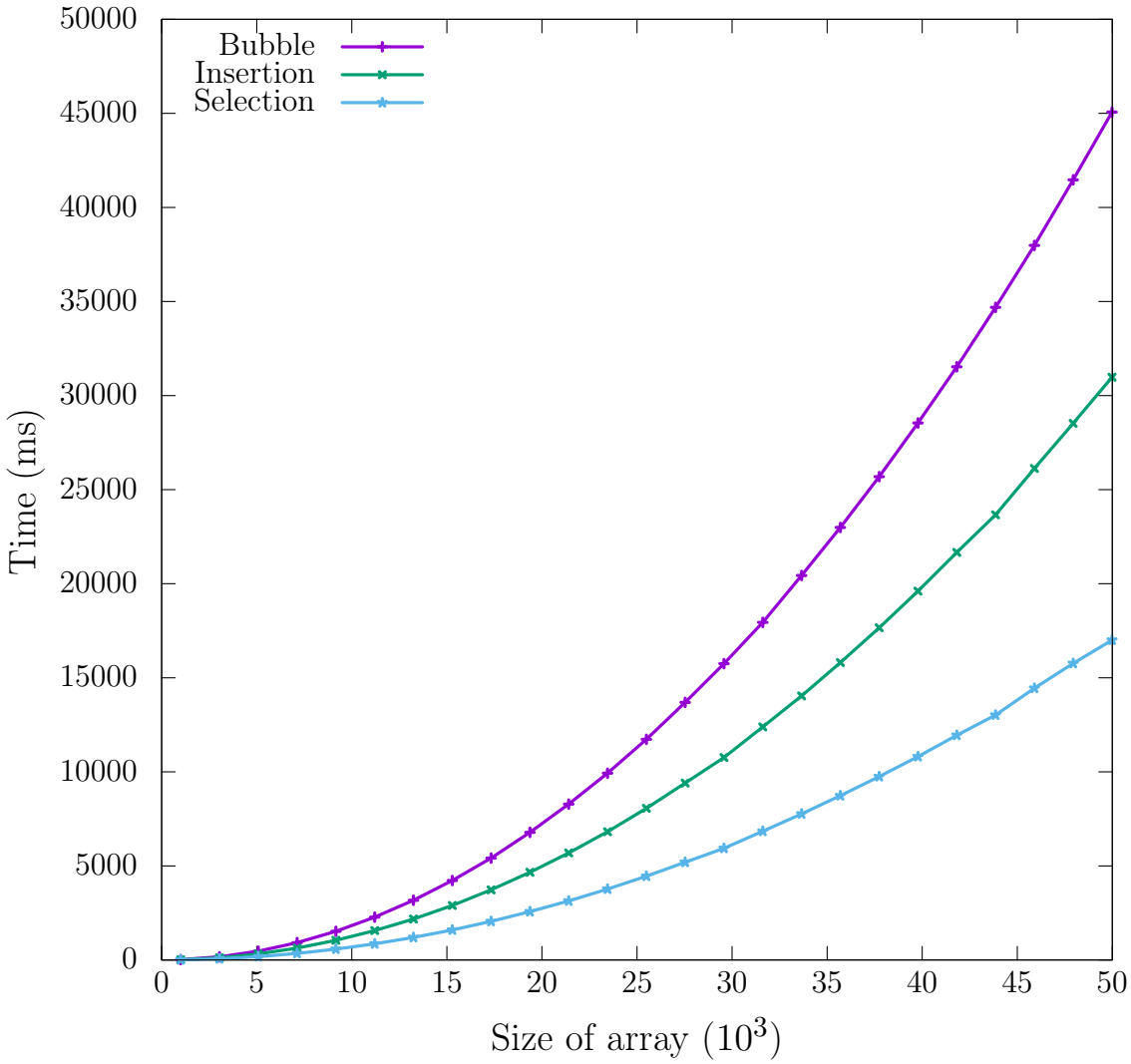

Figura 3: Gráfico dos algoritmos $O(n^2)$ para arranjo não crescente

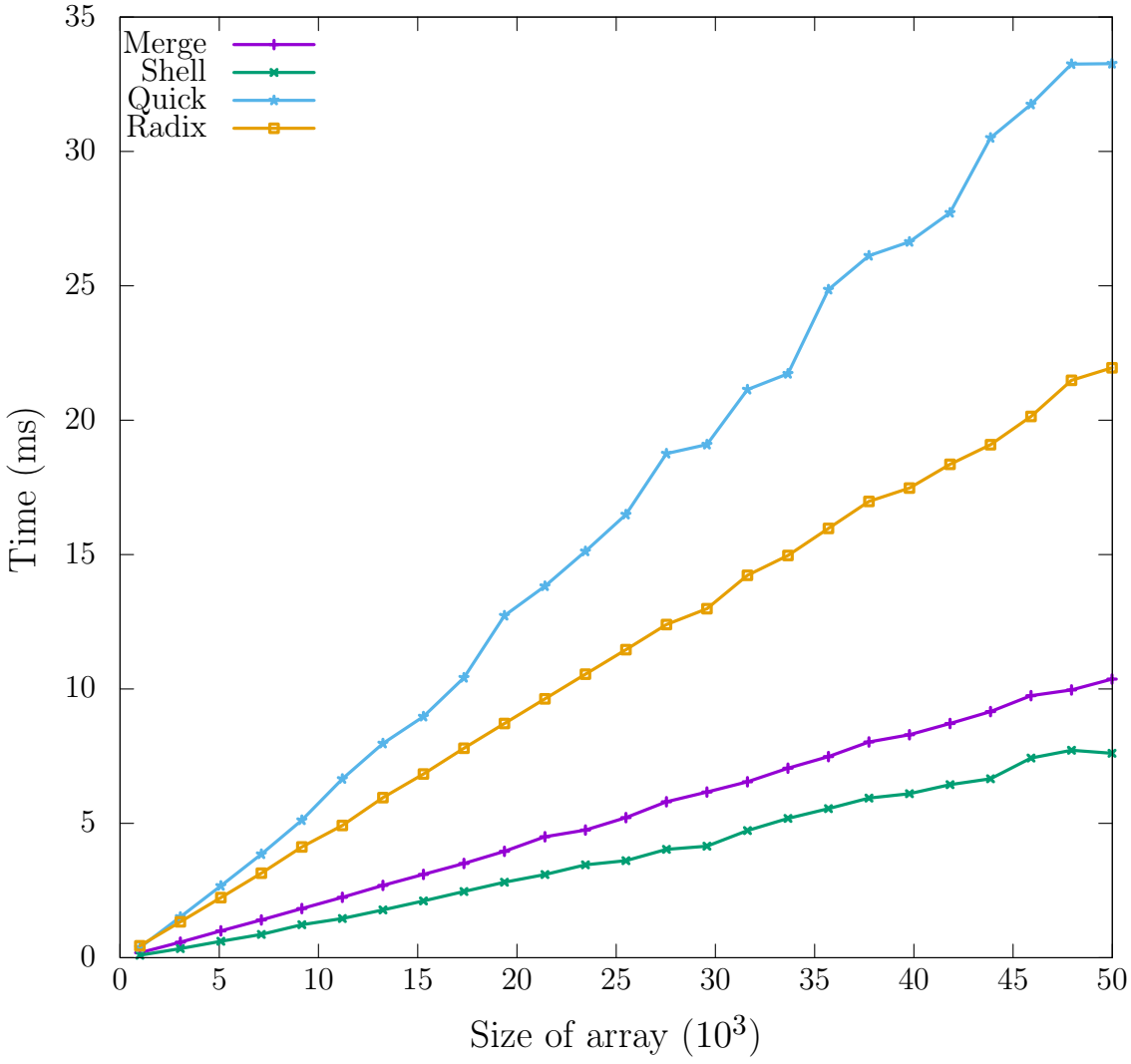

Figura 4: Gráfico dos algoritmos $O(n \log n)$ para arranjo não crescente

3.3 Arranjo em ordem aleatória

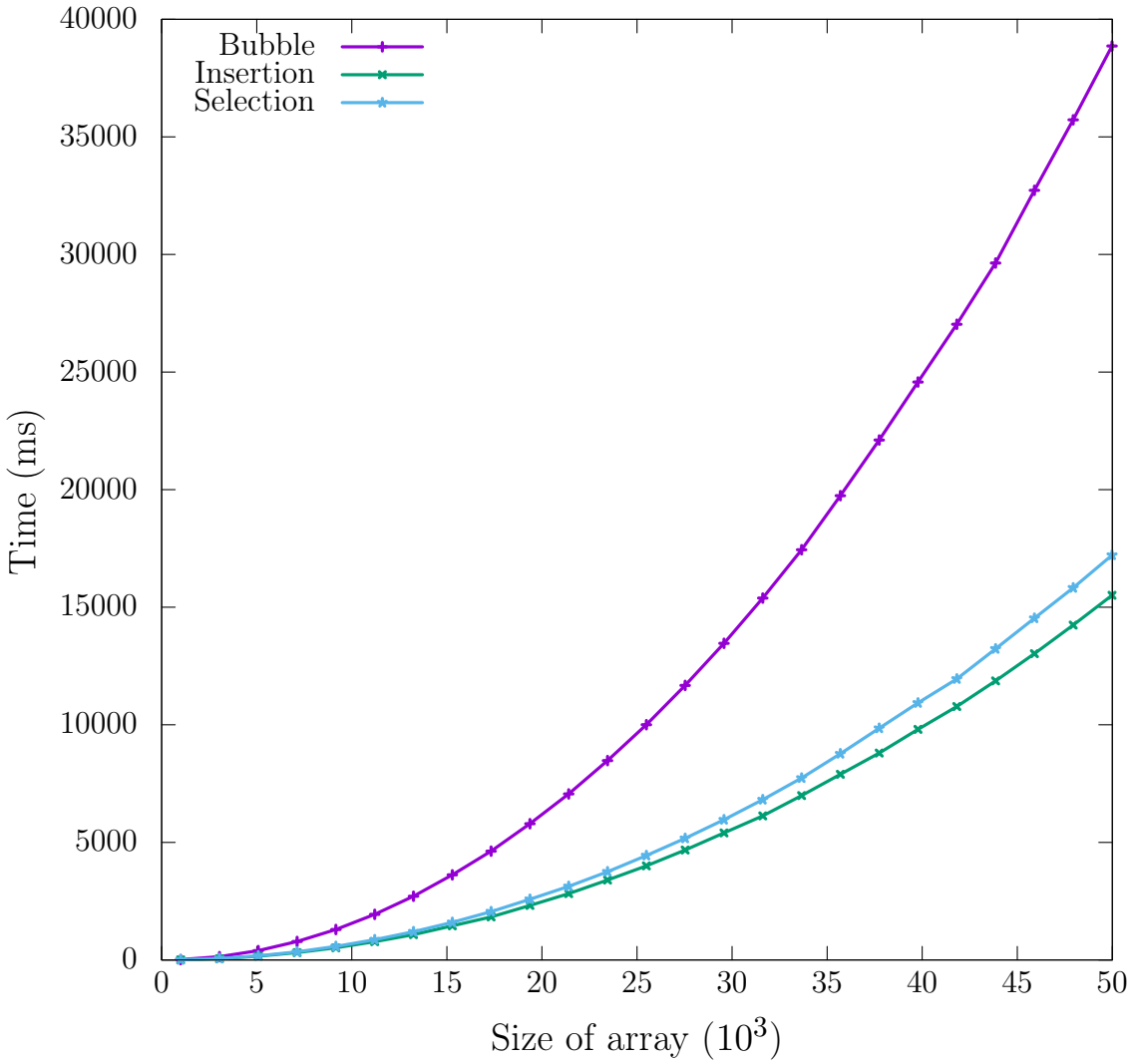

Figura 5: Gráfico dos algoritmos $O(n^2)$ para arranjo aleatório

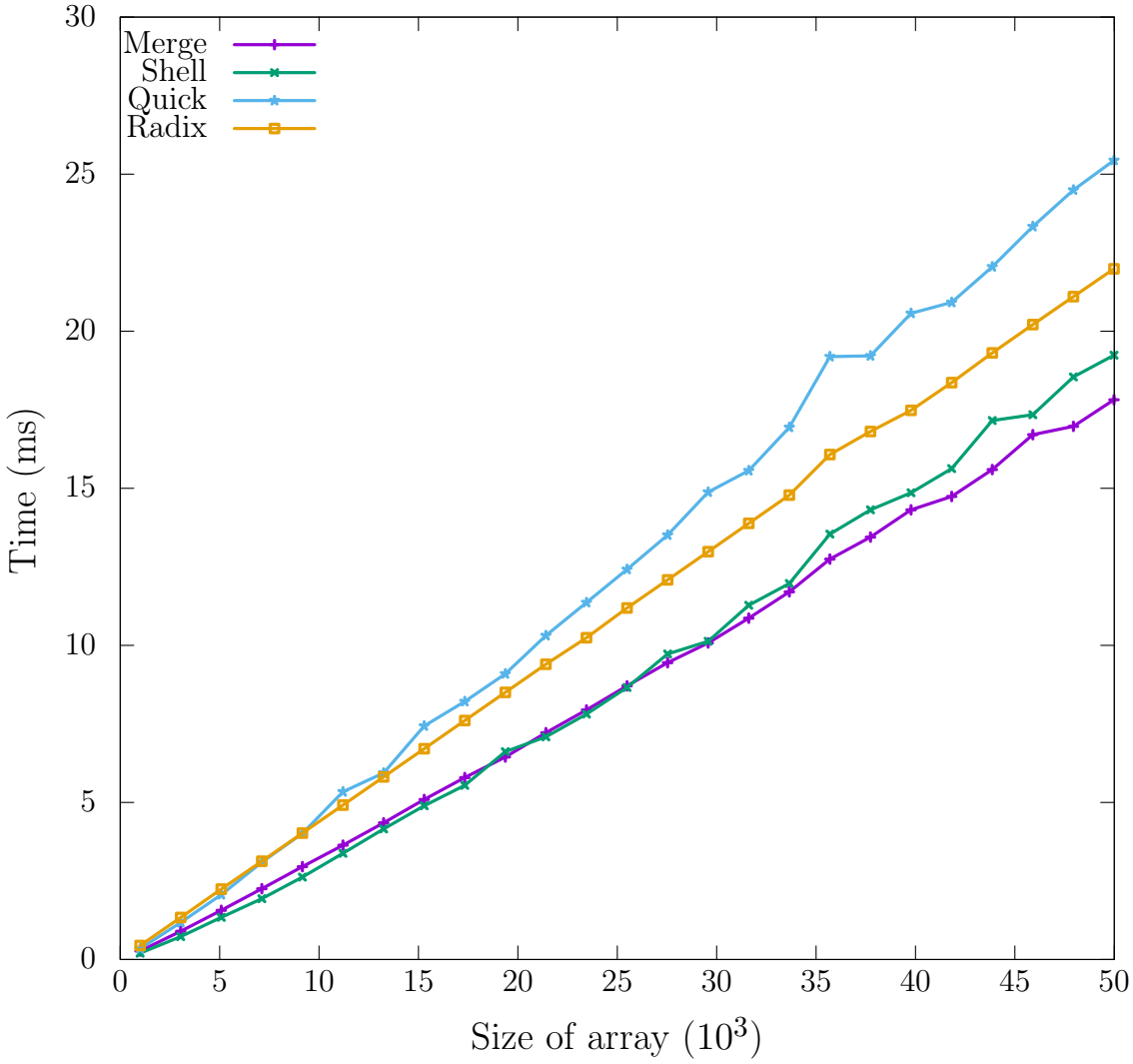

Figura 6: Gráfico dos algoritmos $O(n \log n)$ para arranjo aleatório

3.4 Arranjo com 75% dos seus valores ordenados

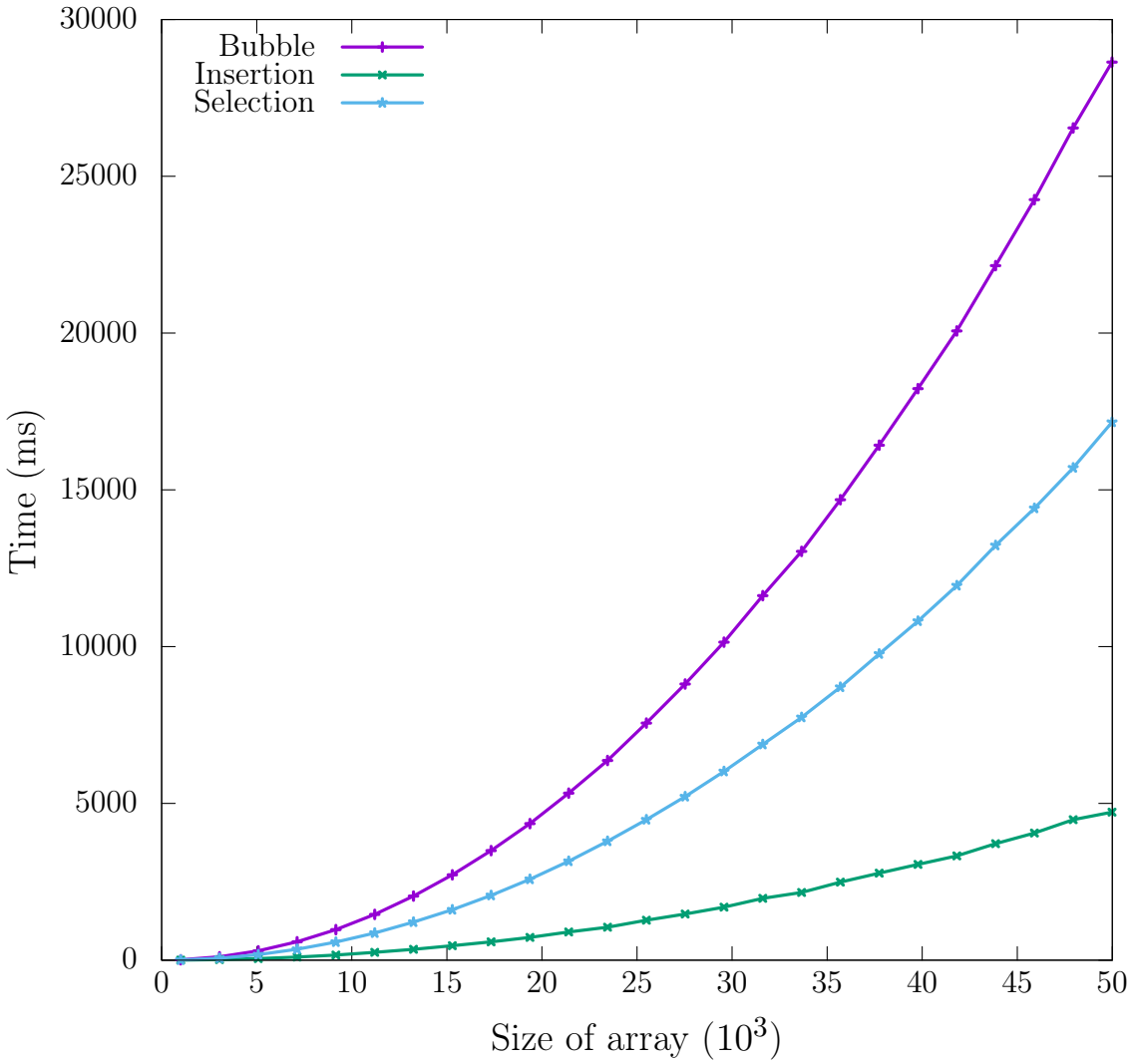

Figura 7: Gráfico dos algoritmos $O(n^2)$ para arranjo 75% ordenado

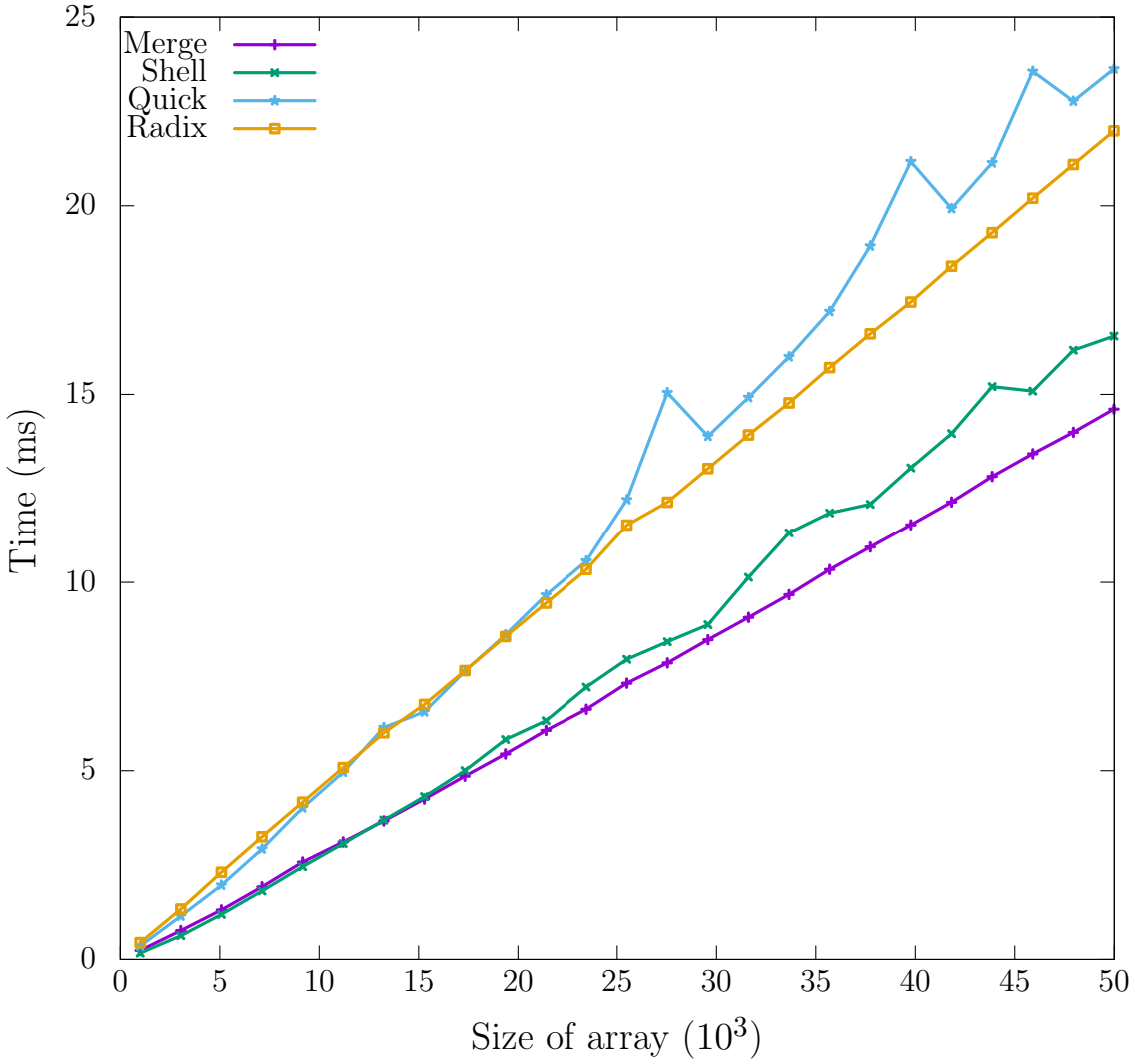

Figura 8: Gráfico dos algoritmos $O(n \log n)$ para arranjo 75% ordenado

3.5 Arranjo com 50% dos seus valores ordenados

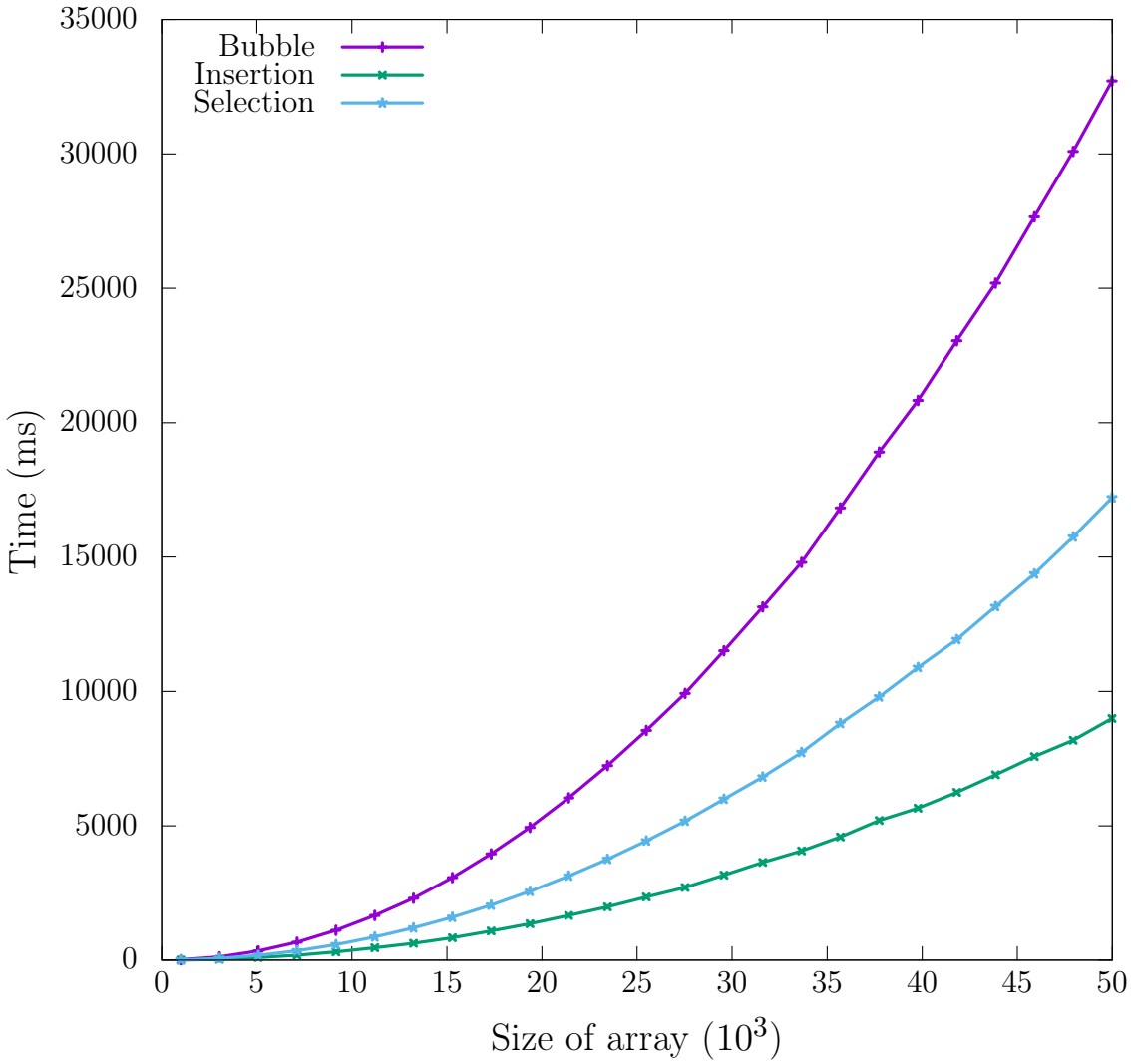

Figura 9: Gráfico dos algoritmos $O(n^2)$ para arranjo 50% ordenado

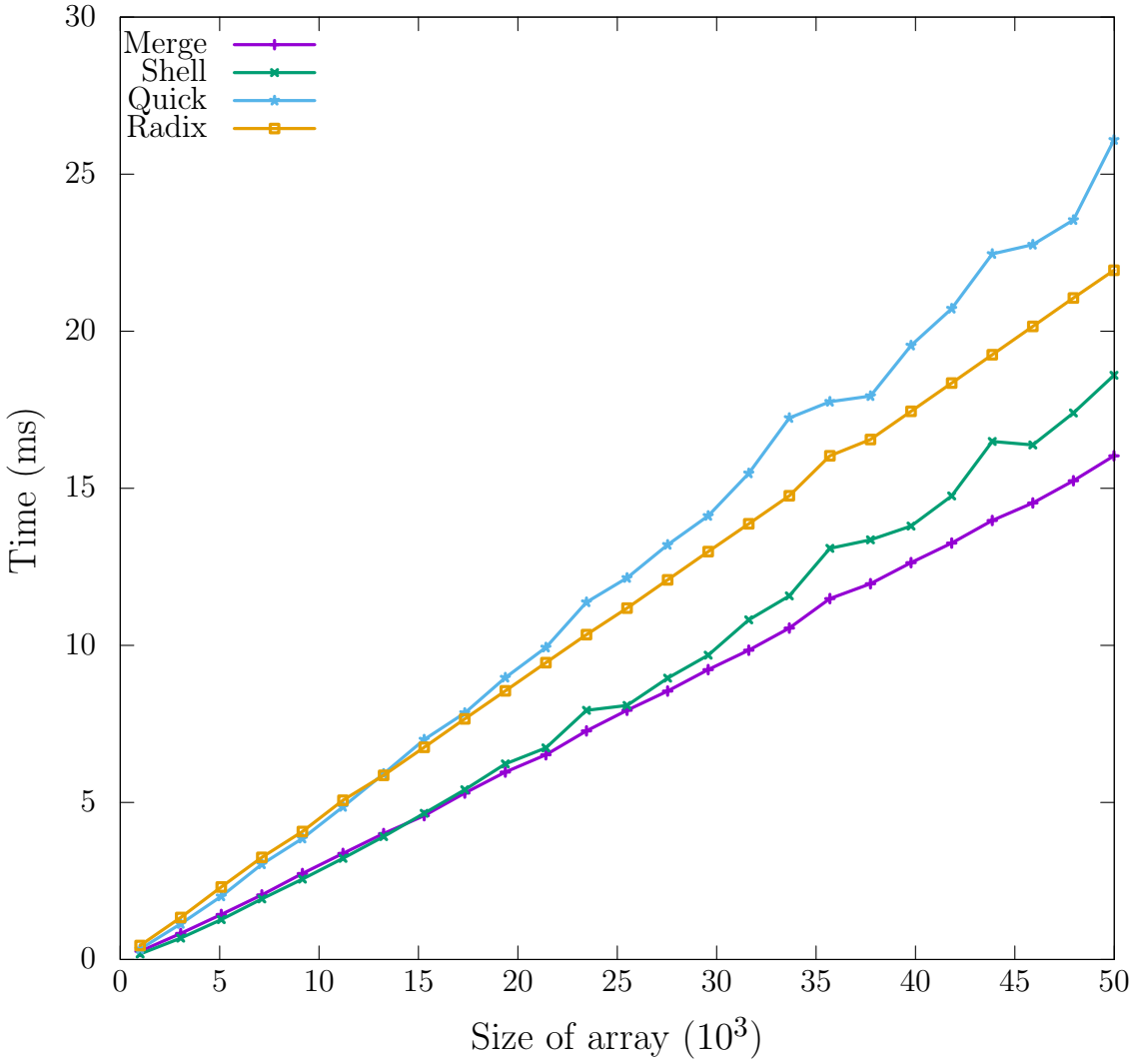

Figura 10: Gráfico dos algoritmos $O(n \log n)$ para arranjo 50% ordenado

3.6 Arranjo com 25% dos seus valores ordenados

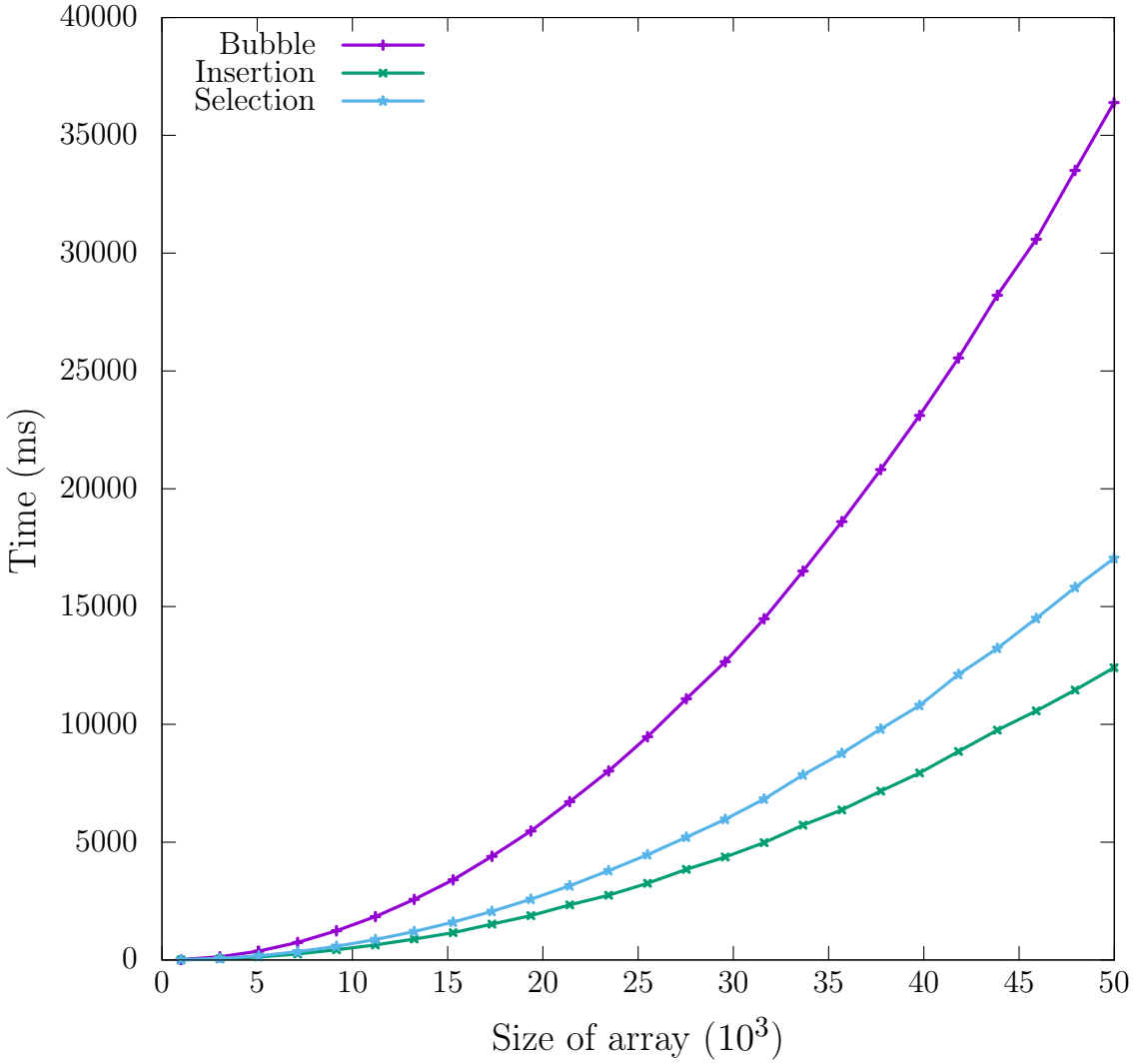

Figura 11: Gráfico dos algoritmos $O(n^2)$ para arranjo 25% ordenado

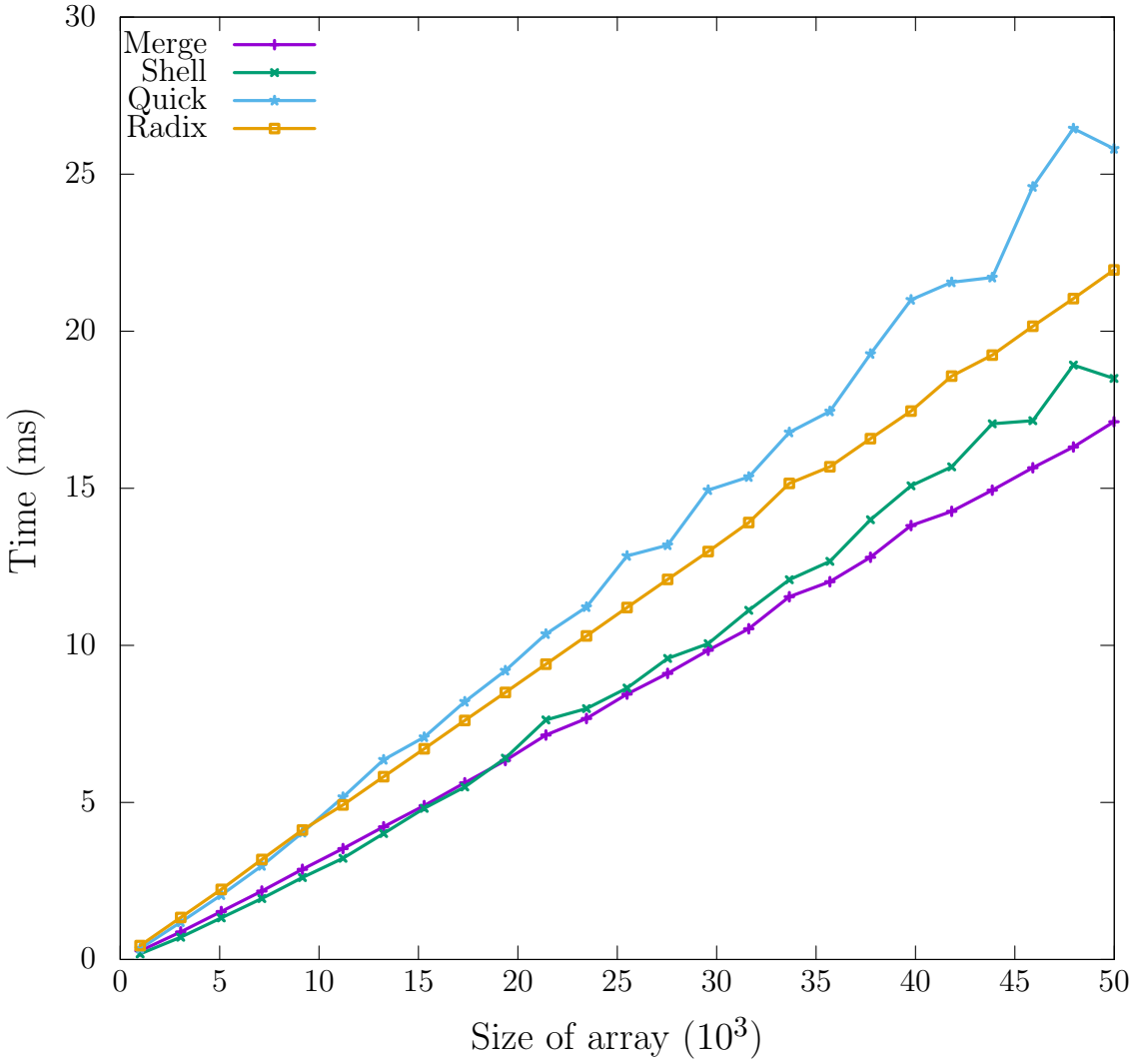

Figura 12: Gráfico dos algoritmos $O(n \log n)$ para arranjo 25% ordenado

4 DISCUSSÕES

Por meio dos testes de execução dos algoritmos, foi possível comprovar que existem diferenças significativas no desempenho de diferentes tipos de algoritmos. A depender da estratégia adotada, uma mesma tarefa pode ser executada com uma grande variação de tempo, principalmente para grandes volumes de dados. Portanto, ficou nítido que um dos possíveis caminhos para economizar recursos computacionais consumidos por softwares é realizar uma boa seleção dos algoritmos a serem adotados.

4.1 Algoritmos recomendados para cada cenário

No cenário onde todos os elementos de um determinado contêiner de dados estão ordenados de forma não decrescente, o algoritmo mais eficiente é o *bubble sort*. Entretanto, não faz sentido escolher um algoritmo de ordenação considerando que o arranjo já está ordenado, ainda mais considerando que o *bubble sort* teve o pior desempenho em todos os outros cenários.

No cenário onde o arranjo estava em ordem não crescente, o *shell sort* é o algoritmo recomendado pois ele foi o mais rápido durante toda a execução e pelo ângulo da reta parece que ele vai continuar sendo o melhor para tamanhos maiores. Além disso, esse algoritmo também teve ótimo desempenho nos outros cenários, perdendo apenas para o *merge sort* quando o arranjo se torna muito grande.

Nos demais cenários, tivemos resultados parecidos: o *shell sort* começa com o melhor desempenho, mas é ultrapassado pelo *merge sort* em tamanhos maiores. Logo, para arranjos de até 20.000 posições é recomendado o *shell sort* e para arranjos, maiores que isso o *merge sort* é o mais recomendado.

Também é importante mencionar que em casos onde se deseja ordenar arranjo de números pequenos, recomenda-se utilizar o *radix sort*. Mesmo ele não tendo sido o mais eficiente em nossa análise, ele ainda tem um ótimo desempenho e sabemos que o tempo

de execução dele depende do tamanho dos números nos arranjos, ou seja, ele ficará ainda melhor quando se deseja ordenar números pequenos.

Em relação a hipótese de que o *quick sort* é, na prática, mais rápido que o *merge sort*, a nossa análise não conseguiu comprovar isso. Em todos os cenários, o *quick sort* teve desempenho pior que o *merge sort*.

4.2 Análise matemática dos algoritmos

Para o algoritmo *bubble sort*, foi encontrada a seguinte função matemática para o seu comportamento: $f(n) = 15.5204 \cdot n - 37.166$

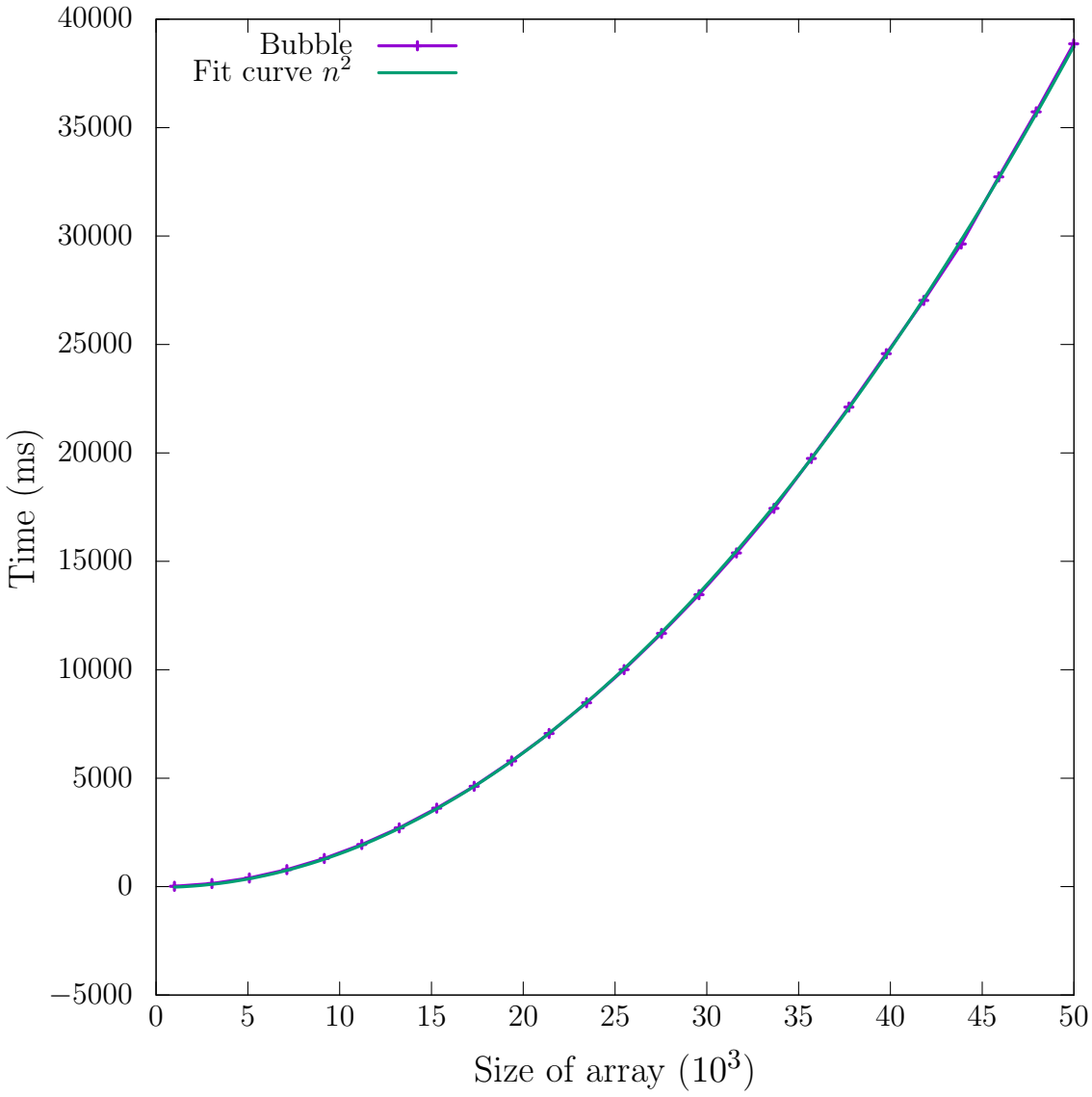

Figura 13: Função matemática: Bubble Sort

Para o algoritmo *insertion sort*, encontramos a seguinte função que corresponde a complexidade do algoritmo:: $f(n) = 6.19065 \cdot n - 8.87531$

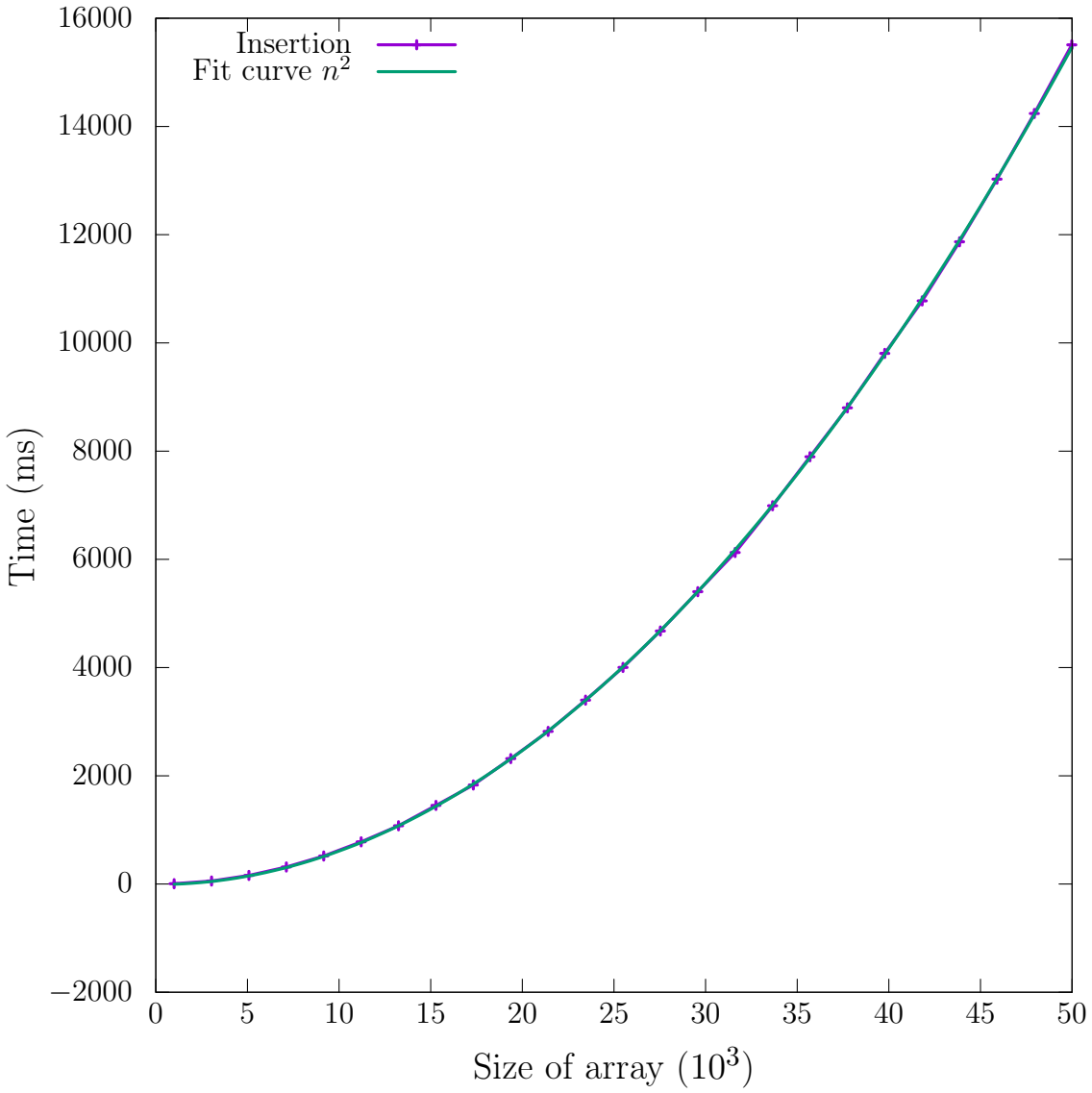

Figura 14: Função matemática: Insertion Sort

Para o algoritmo *select sort*, encontramos a seguinte função que corresponde a complexidade do algoritmo:: $f(n) = 6.88754 \cdot n - 15.6854$

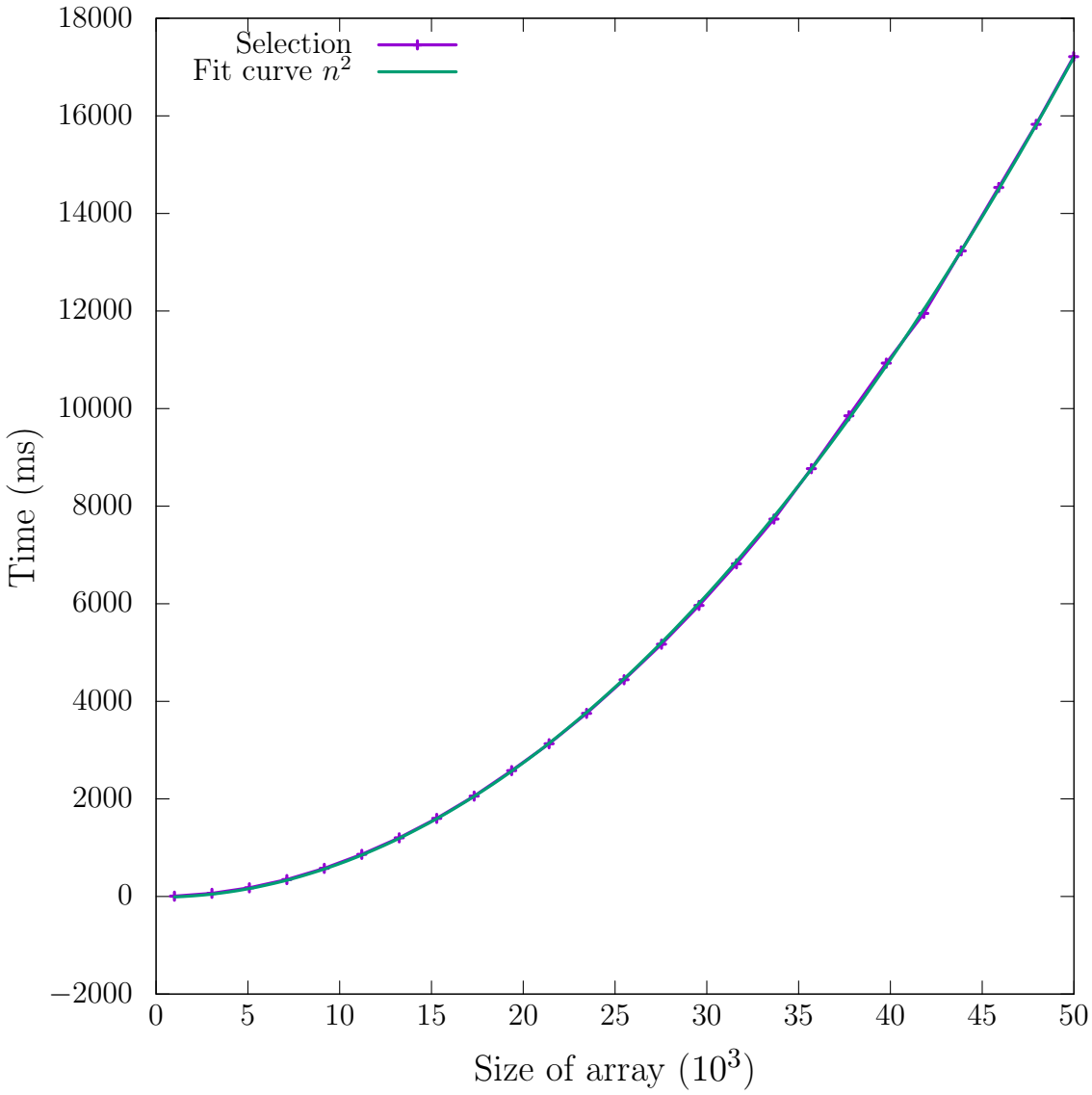

Figura 15: Função matemática: Select Sort

Para o algoritmo *merge sort*, encontramos a seguinte função que corresponde a complexidade do algoritmo: $f(n) = 0.0883011 \cdot n \cdot \log n + 1.1394$.

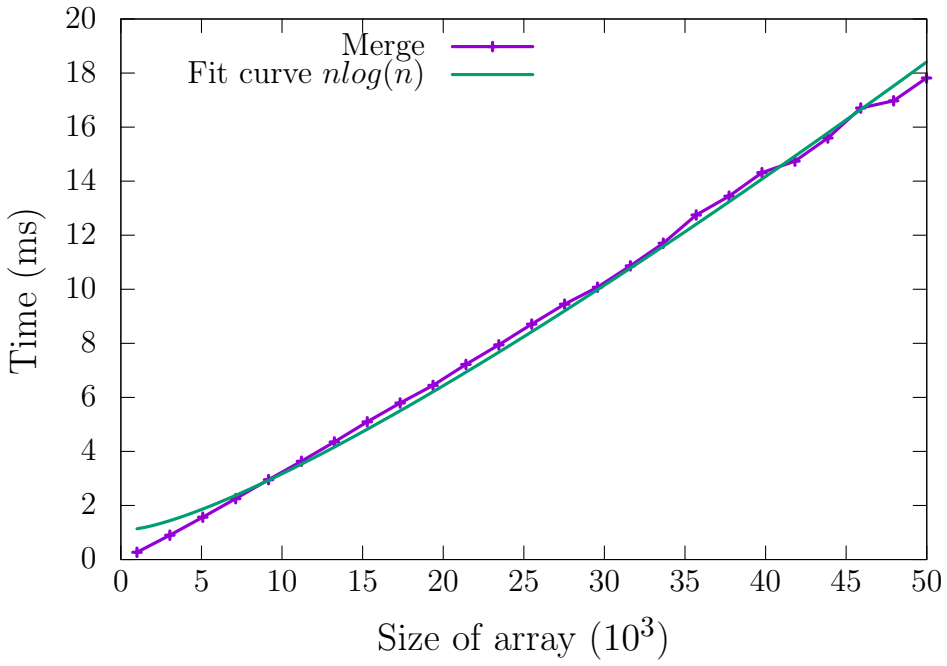

Figura 16: Função matemática: Merge Sort

Para o algoritmo *shell sort*, encontramos a seguinte função que corresponde a complexidade do algoritmo: $f(n) = 0.0970437 \cdot n \cdot \log n + 0.66027$.

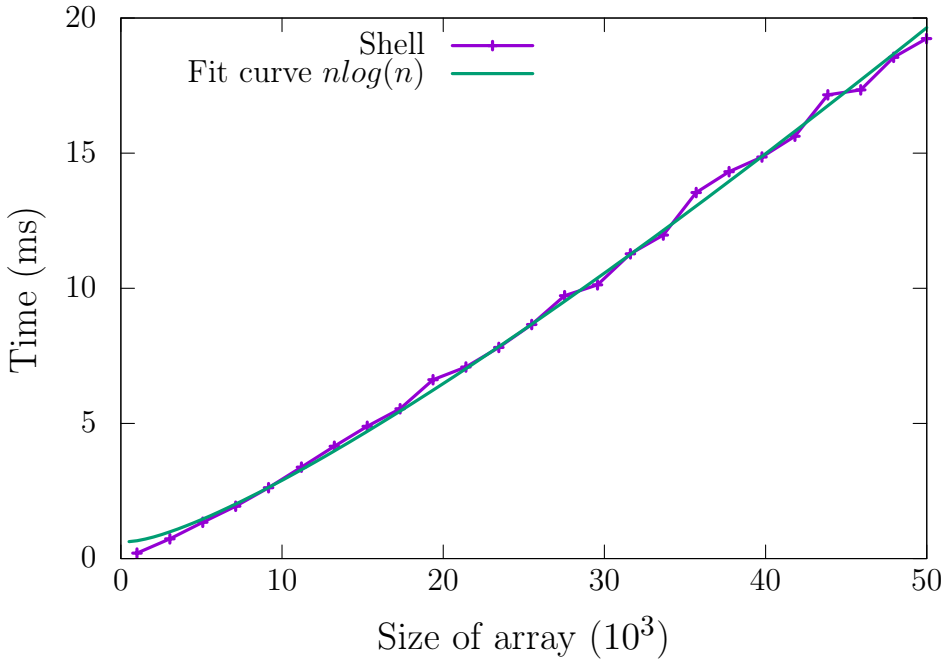

Figura 17: Função matemática: Shell Sort

Para o algoritmo *quick sort*, encontramos a seguinte função que corresponde a complexidade do algoritmo: $f(n) = 0.126833 \cdot n \cdot \log n + 1.58303$.

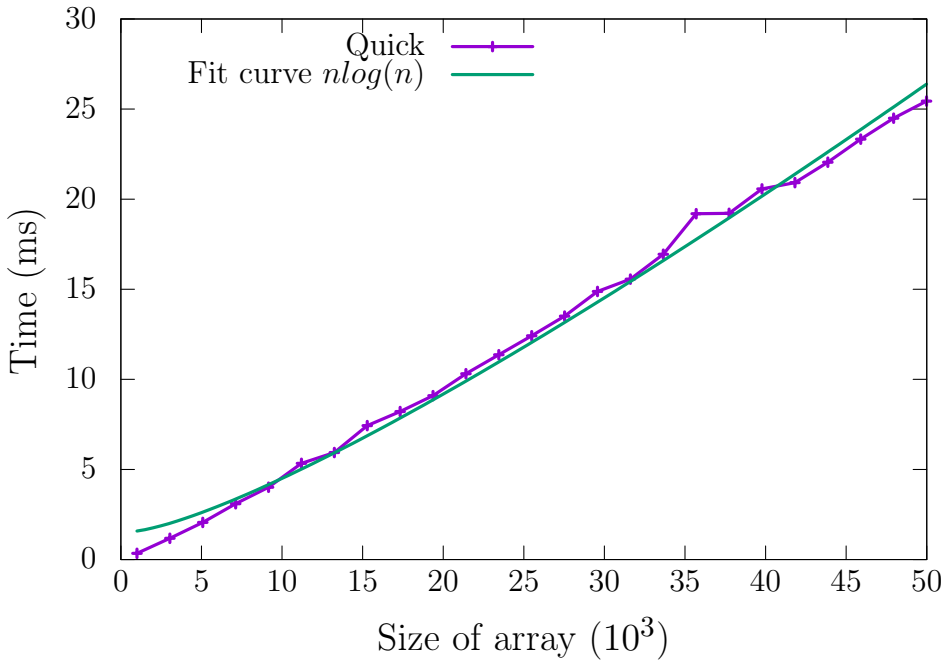

Figura 18: Função matemática: Quick Sort

5 CONCLUSÃO

Considerando os resultados obtidos neste trabalho a partir da realização da análise de comlexidade dos seguintes algoritmos de ordenação: o *Insertion Sort*, *Selection Sort*, *Bubble Sort*, *Shell Sort*, *Quick Sort*, *Merge Sort* e *Radix Sort* (LSD), ficou evidente que, por apresentar melhor eficiência em um quantitativo maior de casos durante os testes de execução, o algoritmo Merge Sort seria o selecinado e indicado para a tarefa de ordenação em um arranjo, já que ele apresentou melhor desempenho na maioria dos casos.

REFERÊNCIAS

CORMEN, Thomas H.; LEISERSON, Chales E.; STEIN, Clifford. **Algoritmos: Teoria E Prática**. [S.l.: s.n.], 2002. ISBN 9788535209266.