Análise de Algoritmos de Ordenação: Comparativo de Desempenho em C

Alexandre Cesar de Souza Rangel[1]

[1]Departamento de Computação – Universidade Federal do Espírito Santo (UFES)
Alegre – ES – Brasil

alexandre.rangel@edu.ufes.br

Abstract. *This work presents the implementation and comparative performance analysis of 14 sorting algorithms, classified into simple (Bubble, Insertion, Selection), efficient (Quick Sort, Merge Sort, Heap Sort, Shell Sort), and linear (Bucket Sort, Radix Sort) methods. The tests were performed in C language, using datasets of 10,000, 100,000, and 500,000 elements in random, increasing, and decreasing scenarios. Experimental results, obtained on an AMD Ryzen 9 processor, demonstrate the practical infeasibility in real-world scenarios of quadratic complexity algorithms ($O(n^2)$) for large data volumes and the superior efficiency of $O(n \log n)$ algorithms.*

Resumo. *Este trabalho apresenta a implementação e a análise comparativa de desempenho de 14 algoritmos de ordenação, divididos em métodos simples (Bolha, Inserção, Seleção), eficientes (Quick Sort, Merge Sort, Heap Sort, Shell Sort) e lineares (Bucket Sort, Radix Sort). Os testes foram realizados em linguagem C, utilizando vetores de 10.000, 100.000 e 500.000 elementos em cenários aleatórios, crescentes e decrescentes. Os resultados experimentais, obtidos em um processador AMD Ryzen 9, demonstram a inviabilidade prática de algoritmos de complexidade quadrática ($O(n^2)$) para grandes volumes de dados e a eficiência superior dos algoritmos $O(n \log n)$.*

1. Introdução

A ordenação é um dos problemas centrais na Ciência da Computação, servindo como bloco construtivo para sistemas complexos, desde a otimização de consultas em bancos de dados relacionais até o pré-processamento para algoritmos de busca e compressão. Embora a teoria da complexidade computacional ofereça limites assintóticos claros para o desempenho — classificando algoritmos em ordens como $O(n^2)$ ou $O(n \log n)$ — a performance prática é frequentemente influenciada por fatores arquiteturais do hardware moderno, como hierarquia de memória, predição de desvio e localidade de referência [Cormen et al. 2012].

Em cenários de *Big Data*, a escolha incorreta de um algoritmo de ordenação pode inviabilizar sistemas. A discrepância entre um algoritmo quadrático e um linear-logarítmico, quando aplicada a grandes volumes de dados, transcende a simples espera do usuário, impactando custos de processamento em nuvem e consumo energético. Além disso, características dos dados de entrada — como vetores parcialmente ordenados ou com muitas chaves duplicadas — podem alterar drasticamente o comportamento de algoritmos adaptativos, como o *Insertion Sort*, ou degradar algoritmos sensíveis, como o *Quick Sort* mal implementado [Ziviani 2010].

Este trabalho apresenta uma análise empírica rigorosa de 14 algoritmos de ordenação, implementados na linguagem C, visando isolar e quantificar o impacto da complexidade algorítmica versus otimizações de hardware. Diferente de abordagens puramente teóricas, este estudo utiliza um processador de alto desempenho (AMD Ryzen 9) para verificar se a "força bruta"computacional é capaz de mitigar a ineficiência de algoritmos $O(n^2)$ em grandes entradas.

Os objetivos específicos deste estudo são:

- Implementar e comparar o tempo de execução (CPU Time) de algoritmos de três classes de complexidade: Quadrática, Logarítmica e Linear;
- Avaliar o impacto da escolha do pivô no algoritmo *Quick Sort* (Fim vs. Mediana de Três);
- Analisar a escalabilidade dos algoritmos de distribuição (*Bucket* e *Radix*) comparados aos baseados em comparação;
- Determinar o ponto de ruptura onde algoritmos elementares tornam-se proibitivos para uso prático.

2. Algoritmos Avaliados

Este estudo contempla 14 implementações distintas, categorizadas por sua estratégia de ordenação e complexidade assintótica.

2.1. Algoritmos Quadráticos ($O(n^2)$)

Métodos simples, geralmente ineficientes para grandes entradas, mas úteis para fins didáticos ou pequenas estruturas.

- **Bubble Sort:** Percorre o vetor repetidamente, trocando elementos adjacentes fora de ordem. Avaliou-se a versão clássica e uma otimizada com flag de parada antecipada.
- **Selection Sort:** Seleciona o menor elemento do subvetor não ordenado e o troca com a posição atual. Realiza o mínimo de trocas ($O(n)$), mas mantém comparações $O(n^2)$.
- **Insertion Sort:** Constrói a ordenação final um item de cada vez. É adaptativo, aproximando-se de $O(n)$ em vetores quase ordenados.

2.2. Algoritmos Eficientes ($O(n \log n)$)

Utilizam estratégias de "divisão e conquista"ou estruturas de dados complexas para reduzir o número de comparações.

- **Shell Sort:** Generalização do Insertion Sort que permite trocas de elementos distantes (baseado em uma sequência de lacunas), reduzindo a desordem rapidamente antes da ordenação fina.
- **Heap Sort:** Utiliza uma estrutura de Heap Binário (árvore) para extrair o maior elemento repetidamente. Garante complexidade $O(n \log n)$ no pior caso e não requer memória auxiliar ($O(1)$).
- **Merge Sort:** Divide o vetor recursivamente ao meio e combina (intercala) os subvetores ordenados. É estável, porém demanda espaço auxiliar proporcional a N.
- **Quick Sort:** Escolhe um elemento como pivô e particiona o vetor de forma que menores fiquem à esquerda e maiores à direita. Foram implementadas três estratégias de escolha de pivô para mitigar o pior caso $O(n^2)$: (1) Último elemento, (2) Elemento central e (3) Mediana de três.

2.3. Algoritmos de Distribuição Linear ($O(n)$)

Algoritmos que não utilizam comparação direta entre chaves, explorando propriedades numéricas dos dados.

- **Bucket Sort:** Distribui os elementos em "baldes"baseados em intervalos de valores, ordenando-os individualmente. Eficiente para dados uniformemente distribuídos.
- **Radix Sort:** Ordena processando os dígitos individuais dos números (do menos significativo para o mais significativo), agrupando chaves com o mesmo dígito na mesma posição.

3. Resultados e Discussão

Esta seção detalha a configuração experimental e discute os dados obtidos. A análise principal foca no cenário crítico de $N = 500.000$ elementos. **Os gráficos e tabelas complementares referentes aos cenários de $N = 10.000$ e $N = 100.000$ encontram-se disponíveis no Apêndice deste trabalho.**

3.1. Configuração Experimental

Para garantir a reprodutibilidade e a precisão da análise, os algoritmos foram implementados em C e instrumentados para capturar três indicadores fundamentais de desempenho:

1. **Tempo de Execução (CPU Time):** Medido através da função `clock()` da biblioteca `time.h`, isolando o tempo de processamento de interrupções de I/O.
2. **Número de Comparações:** Contador incremental acionado a cada verificação condicional entre chaves (ex: `if (v[i] > v[j])`). Essencial para validar a complexidade teórica.
3. **Número de Movimentações (Trocas):** Contabiliza quantas vezes os registros foram copiados ou trocados de posição na memória. Métrica crítica para avaliar o custo em cenários onde a escrita é custosa.

Os códigos foram compilados com GCC. Os testes foram executados em um ambiente de alto desempenho para minimizar gargalos de hardware: processador **AMD Ryzen 9 7940HS** (8 núcleos, 16 threads, Boost até 5.2 GHz), 32 GB de memória RAM DDR5 e sistema operacional Windows 11.

Os datasets consistem em vetores de inteiros de 32 bits gerados em três disposições distintas, seguindo a metodologia de análise experimental proposta por [Ziviani 2010] para avaliar o comportamento dos algoritmos em seus limites assintóticos:

1. **Aleatório:** Cenário de caso médio, onde os elementos não possuem padrão prévio, representando a situação mais comum em aplicações reais.
2. **Crescente:** Cenário que explora o *Melhor Caso* de algoritmos adaptativos (como o *Insertion Sort*), mas que pode induzir o *Pior Caso* em implementações ingênuas do *Quick Sort*.
3. **Decrescente:** Cenário tipicamente utilizado para simular o *Pior Caso* em algoritmos elementares, exigindo o número máximo de movimentações.

3.2. Análise: Algoritmos Quadráticos e Custo de Movimentação

Como previsto pela teoria, os algoritmos $O(n^2)$ tornaram-se inviáveis para grandes volumes de dados, com o *Bubble Sort* exigindo mais de 8 minutos para concluir a ordenação aleatória. No entanto, a análise das métricas secundárias revela nuances importantes:

- **Selection Sort vs. Bubble Sort:** Embora ambos possuam complexidade quadrática de comparações, o *Selection Sort* apresentou desempenho superior. A razão reside no número de trocas (swaps): enquanto o *Bubble Sort* realiza $O(n^2)$ escritas na memória, o *Selection Sort* realiza apenas $O(n)$ trocas. Em arquiteturas onde a escrita em memória é uma operação custosa, o Selection Sort mostra-se uma alternativa muito mais eficiente, apesar da mesma complexidade assintótica de tempo.
- **Adaptabilidade:** No cenário **Crescente**, o *Insertion Sort* foi virtualmente instantâneo (milissegundos), validando sua característica adaptativa ($N - 1$ comparações e zero trocas), superando até o Quick Sort.

3.3. Análise: Algoritmos Eficientes ($O(n \log n)$) e Hierarquia de Memória

Os algoritmos eficientes mantiveram o tempo abaixo de 300ms para 500.000 elementos.

- **Movimentação no Merge Sort:** Vale ressaltar que, no *Merge Sort*, a métrica de ”trocas”contabiliza as movimentações de dados (cópias) entre o vetor original e o vetor auxiliar durante o processo de intercalação. Embora não sejam trocas diretas (*swaps*) como no Bubble Sort, essas operações representam o custo real de reordenação do vetor na memória.
- **Heap Sort vs. Quick Sort:** Teoricamente, o *Heap Sort* é vantajoso por garantir $O(n \log n)$ no pior caso e usar memória constante $O(1)$. Porém, na prática, ele mostrou-se consistentemente mais lento que o *Quick Sort*. Esse comportamento **pode estar relacionado** ao padrão de acesso à memória do Heap (saltos de índice $i \to 2i$), que tende a apresentar menor localidade de referência espacial, causando falhas constantes na Cache do processador Ryzen. O Quick Sort, ao varrer o vetor linearmente durante o particionamento, aproveita melhor a pré-busca (prefetching) da CPU.
- **Sensibilidade do Quick Sort:** O experimento confirmou a fragilidade da implementação clássica (Pivô no Fim). No cenário Crescente, o algoritmo degradou para $O(n^2)$, assemelhando-se ao desempenho do Bubble Sort. A estratégia de “Mediana de Três” corrigiu eficazmente este problema, mantendo a complexidade $O(n \log n)$ e provando ser indispensável para implementações de biblioteca padrão.

3.4. Análise: Algoritmos Lineares

O *Bucket Sort* e o *Radix Sort* superaram todos os algoritmos baseados em comparação, operando em tempo linear $O(n)$.

- **Custo Escondido do Radix Sort:** Apesar da complexidade linear, o Radix Sort executa múltiplas passagens sobre os dados (uma por dígito), o que aumenta o número de movimentações e acessos à memória. Esse fator explica seu desempenho inferior ao Bucket Sort em alguns cenários, mesmo mantendo O(n).

- **Dependendência da Distribuição do Bucket Sort:** O desempenho do Bucket Sort depende fortemente da uniformidade da distribuição dos dados. Em distribuições adversas, o custo interno de ordenação dos baldes pode degradar o desempenho, aproximando-o de algoritmos comparativos.

3.5. Síntese do Desempenho

A Figura 1 ilustra o comparativo final. Fica demonstrado que a otimização algorítmica é o fator dominante: a diferença entre um algoritmo linear-logarítmico e um quadrático é de várias ordens de magnitude, algo que nem a frequência de 5.2 GHz do processador foi capaz de mitigar.

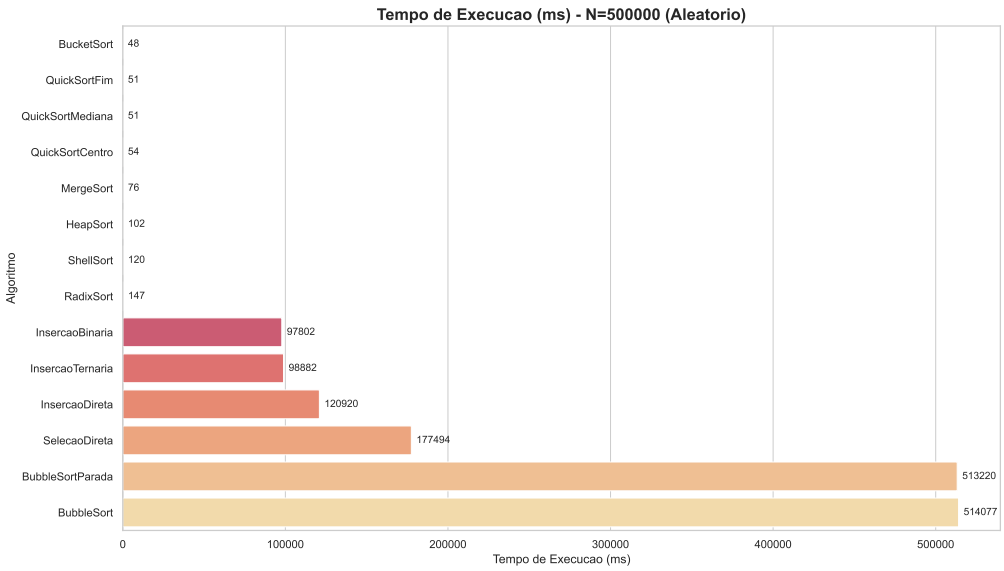

Figura 1. Tempo de execução para 500.000 elementos (Escala Logarítmica).

Tabela 1. Desempenho Experimental: N = 500.000 Elementos

	Crescente			Decrescente			Aleatório		
Algoritmo	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**
Algoritmos Quadráticos									
Bubble Sort	197916	1.2E11	0	397089	1.2E11	1.2E11	514077	1.2E11	6.2E10
Bubble (Parada)	1	5.0E5	0	429657	1.2E11	1.2E11	513220	1.2E11	6.2E10
Selection Sort	179815	1.2E11	0	193266	1.2E11	2.5E5	177494	1.2E11	5.0E5
Inserção Direta	1	5.0E5	5.0E5	249957	1.2E11	1.2E11	120920	6.2E10	6.2E10
Inserção Binária	19	9.0E6	5.0E5	198974	8.5E6	1.2E11	97802	8.7E6	6.2E10
Inserção Ternária	32	1.1E7	5.0E5	198762	5.6E6	1.2E11	98882	9.3E6	6.2E10
Algoritmos Eficientes									
Shell Sort	28	8.5E6	8.5E6	40	1.2E7	1.3E7	120	2.8E7	2.8E7
Heap Sort	78	1.8E7	9.4E6	74	1.7E7	8.7E6	102	1.7E7	9.0E6
Merge Sort	44	4.8E6	1.9E7	38	4.7E6	1.9E7	76	8.8E6	1.9E7
Quick (Fim)	397624	1.2E11	1.2E11	284000	1.2E11	6.2E10	51	1.2E7	5.8E6
Quick (Centro)	24	8.5E6	4.7E6	28	8.7E6	5.0E6	54	1.2E7	6.8E6
Quick (Mediana)	26	9.0E6	4.7E6	51	1.9E7	1.1E7	51	1.0E7	5.5E6
Algoritmos Lineares									
Bucket Sort	38	1.0E6	1.0E6	39	1.0E6	1.0E6	48	1.2E6	1.3E6
Radix Sort	88	5.0E5	6.0E6	85	5.0E5	6.0E6	147	5.0E5	1.0E7

Nota: Valores em notação científica (E).

4. Conclusão

Os experimentos realizados neste trabalho corroboram a primazia da complexidade assintótica sobre o poder bruto de processamento. Mesmo utilizando uma arquitetura moderna (AMD Ryzen 9 com memória DDR5), algoritmos de ordem quadrática ($O(n^2)$) mostraram-se inviáveis para grandes volumes de dados ($N = 500.000$), atingindo tempos na escala de minutos, enquanto métodos logarítmicos concluíram a tarefa em milissegundos.

A análise detalhada permitiu conclusões específicas além do tempo de execução:

- **Custo de Movimentação:** O *Selection Sort*, apesar de quadrático, provou-se superior ao *Bubble Sort* por minimizar escritas na memória, sendo uma alternativa válida onde a movimentação de registros é custosa.
- **Adaptabilidade:** O *Insertion Sort* confirmou ser a melhor escolha para vetores quase ordenados, superando até mesmo o *Quick Sort* neste nicho específico.
- **Estabilidade e Robustez:** A variação do *Quick Sort* com pivô "Mediana de Três"eliminou o risco de degradação para o pior caso, firmando-se como a escolha mais equilibrada para uso geral. Em contrapartida, o *Heap Sort*, embora teoricamente eficiente, sofreu penalidades de desempenho devido à falta de localidade de referência (impacto no Cache do processador).

Em suma, não existe um "algoritmo perfeito"universal; a escolha ideal depende intrinsecamente do volume de dados, da ordenação prévia da entrada e das restrições de arquitetura (memória vs. processamento).

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos: teoria e prática*. Elsevier Brasil.

Ziviani, N. (2010). *Projeto de algoritmos: com implementações em Pascal e C*. Cengage Learning Editores.

A. Gráficos Complementares: N=10.000

Aqui são apresentados os resultados para o cenário de menor carga.

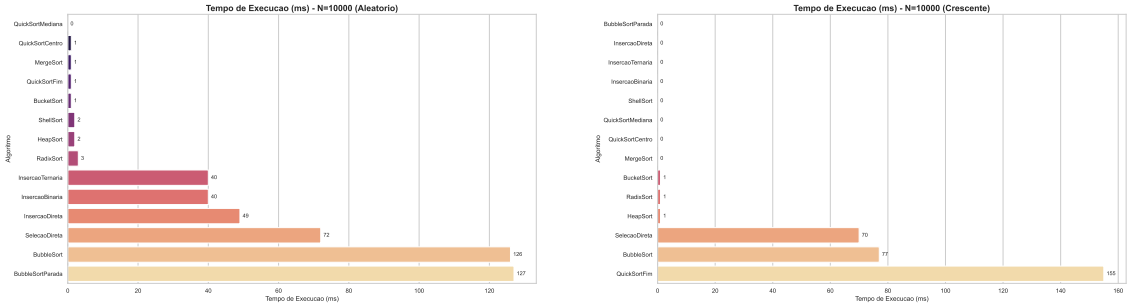

Figura 2. Comparativo de Tempo para N=10.000 (Esq: Aleatório, Dir: Crescente)

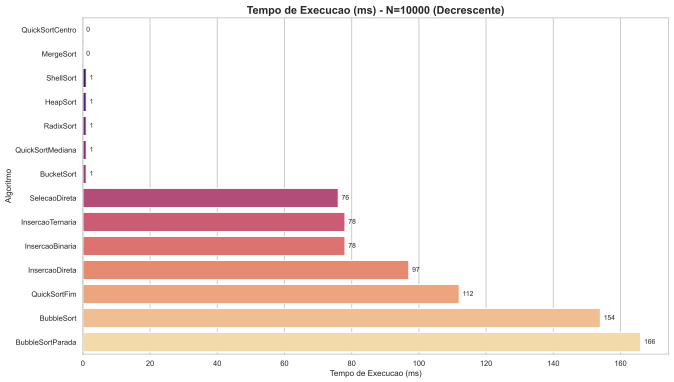

Figura 3. Comparativo de Tempo para N=10.000 (Decrescente)

B. Gráficos Complementares: N=100.000

Aqui observamos a transição de comportamento dos algoritmos.

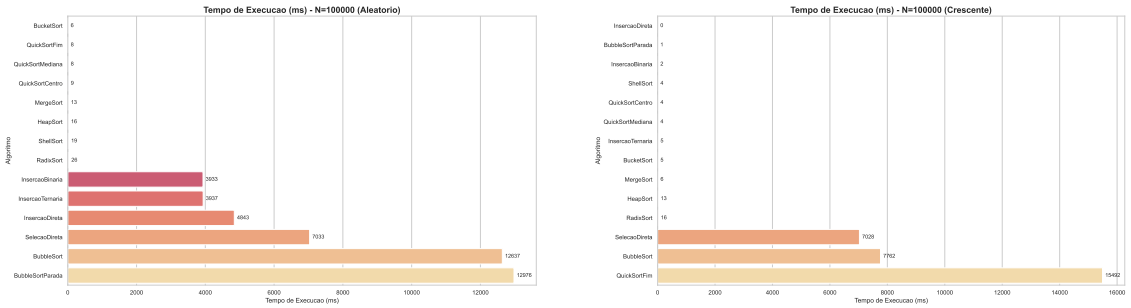

Figura 4. Comparativo de Tempo para N=100.000 (Esq: Aleatório, Dir: Crescente)

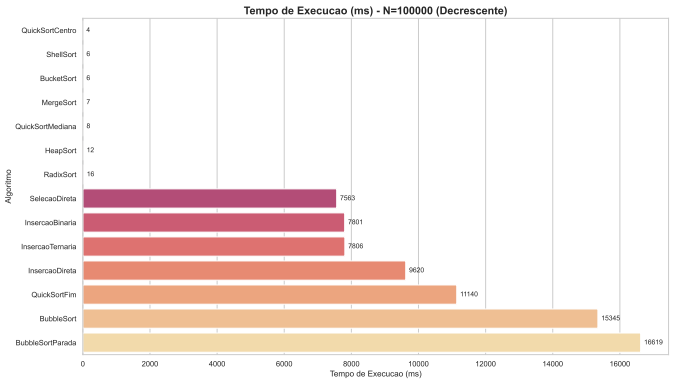

Figura 5. Comparativo de Tempo para N=100.000 (Decrescente)

C. Tabelas de Desempenho Detalhadas

Tabela 2. Desempenho Experimental: N = 10.000 Elementos

	Crescente			Decrescente			Aleatório		
Algoritmo	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**
Algoritmos Quadráticos									
Bubble Sort	77	5.0E7	0	154	5.0E7	5.0E7	126	5.0E7	2.5E7
Bubble (Parada)	0	9.9E3	0	166	5.0E7	5.0E7	127	5.0E7	2.5E7
Selection Sort	70	5.0E7	0	76	5.0E7	5.0E3	72	5.0E7	1.0E4
Inserção Direta	0	9.9E3	9.9E3	97	5.0E7	5.0E7	49	2.5E7	2.5E7
Inserção Binária	0	1.2E5	9.9E3	78	1.1E5	5.0E7	40	1.2E5	2.5E7
Inserção Ternária	0	1.5E5	9.9E3	78	7.5E4	5.0E7	40	1.3E5	2.5E7
Algoritmos Eficientes									
Shell Sort	0	1.2E5	1.2E5	1	1.7E5	1.8E5	2	2.7E5	2.7E5
Heap Sort	1	2.4E5	1.3E5	1	2.3E5	1.2E5	2	2.4E5	1.2E5
Merge Sort	0	6.9E4	2.7E5	0	6.5E4	2.7E5	1	1.2E5	2.7E5
Quick (Fim)	155	5.0E7	5.0E7	112	5.0E7	2.5E7	1	1.5E5	8.6E4
Quick (Centro)	0	1.1E5	6.6E4	0	1.2E5	7.3E4	1	1.6E5	1.0E5
Quick (Mediana)	0	1.3E5	6.8E4	1	2.4E5	1.4E5	0	1.4E5	7.7E4
Algoritmos Lineares									
Bucket Sort	1	2.1E4	2.1E4	1	2.1E4	2.1E4	1	2.5E4	2.6E4
Radix Sort	1	1.0E4	1.0E5	1	1.0E4	1.0E5	3	1.0E4	2.0E5

Tabela 3. Desempenho Experimental: N = 100.000 Elementos

	Crescente			Decrescente			Aleatório		
Algoritmo	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**
Algoritmos Quadráticos									
Bubble Sort	7762	5.0E9	0	15345	5.0E9	5.0E9	12637	5.0E9	2.5E9
Bubble (Parada)	1	1.0E5	0	16619	5.0E9	5.0E9	12976	5.0E9	2.5E9
Selection Sort	7028	5.0E9	0	7563	5.0E9	5.0E4	7033	5.0E9	1.0E5
Inserção Direta	0	1.0E5	1.0E5	9620	5.0E9	5.0E9	4843	2.5E9	2.5E9
Inserção Binária	2	1.6E6	1.0E5	7801	1.5E6	5.0E9	3933	1.5E6	2.5E9
Inserção Ternária	5	1.9E6	1.0E5	7806	9.7E5	5.0E9	3937	1.6E6	2.5E9
Algoritmos Eficientes									
Shell Sort	4	1.5E6	1.5E6	6	2.2E6	2.3E6	19	4.3E6	4.3E6
Heap Sort	13	3.1E6	1.7E6	12	2.9E6	1.5E6	16	3.0E6	1.6E6
Merge Sort	6	8.5E5	3.3E6	7	8.2E5	3.3E6	13	1.5E6	3.3E6
Quick (Fim)	15492	5.0E9	5.0E9	11140	5.0E9	2.5E9	8	2.1E6	1.0E6
Quick (Centro)	4	1.5E6	8.5E5	4	1.5E6	9.0E5	9	2.0E6	1.2E6
Quick (Mediana)	4	1.6E6	8.8E5	8	3.2E6	2.0E6	8	1.8E6	9.4E5
Algoritmos Lineares									
Bucket Sort	5	2.1E5	2.1E5	6	2.1E5	2.1E5	6	2.5E5	2.6E5
Radix Sort	16	1.0E5	1.2E6	16	1.0E5	1.2E6	26	1.0E5	2.0E6