Algoritmos de Ordenação
Análise de Diferentes Entradas

Rafael Cruz M. Nunes [1], **Filipe Mendes O. Rocha** [1]

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

rafael.nunes@edu.ufes.br, filipe.m.rocha@edu.ufes.br

Abstract. *Sorting algorithms vary in efficiency, assessed through comparisons, swaps, and runtime. Techniques like Quick Sort and Merge Sort minimize comparisons, whereas Bubble Sort and Insertion Sort tend to involve more swaps. Efficient algorithms, such as Merge Sort, often exhibit shorter runtime, a critical factor when handling substantial datasets. The choice of algorithm depends on striking a balance among these factors, adapting to dataset characteristics and system performance requirements.*

Resumo. *Os algoritmos de ordenação diferem em eficiência, avaliada por comparações, trocas e tempo de execução. Métodos como Quick Sort e Merge Sort minimizam comparações, enquanto Bubble Sort e Insertion Sort tendem a mais trocas. Algoritmos eficientes, como Merge Sort, geralmente têm menor tempo de execução, crucial ao lidar com conjuntos de dados significativos. A escolha do algoritmo depende do equilíbrio entre esses fatores, adaptando-se às características do conjunto de dados e aos requisitos de desempenho do sistema.*

1. Algoritmos de ordenação

Bubble Sort (Ordenação por Bolha):

Percorre a lista várias vezes, trocando elementos adjacentes se estiverem fora de ordem. Simples, mas ineficiente para grandes conjuntos. Complexidade: $O(n^2)$.

Insertion Sort (Ordenação por Inserção):

Constrói a lista ordenada um elemento por vez, inserindo cada elemento na posição correta. Eficiente para pequenos conjuntos ou listas quase ordenadas. Complexidade: $O(n^2)$.

Insertion Sort Binário:

Utiliza busca binária para encontrar a posição correta de inserção de cada elemento. Reduz comparações em comparação com a inserção direta. Complexidade: O(n log n).

Shell Sort:

Variação do Insertion Sort com comparações espaçadas. Combina eficiência e simplicidade. Complexidade: O(n log n) em média.

Selection Sort (Ordenação por Seleção):

Seleciona iterativamente o menor elemento e o move para a posição correta. Simples, mas ineficiente para grandes conjuntos. Complexidade: O(n^2).

Heap Sort:

Transforma a lista em uma árvore de heap e extrai o maior elemento. Eficiente e estável, não sensível à distribuição inicial dos dados. Complexidade: O(n log n).

Quick Sort:

Escolhe um pivô, particiona a lista e ordena recursivamente as partes. Rápido e eficiente, mas sensível à escolha do pivô. Complexidade: O(n log n) em média.

Merge Sort:

Divide a lista, ordena cada sublista e mescla as sublistas ordenadas. Eficiente e estável, ideal para grandes conjuntos de dados. Complexidade: O(n log n).

Radix Sort:

Ordena os elementos por dígitos, da unidade à casa mais significativa. Eficiente para dados com representação decimal. Complexidade: O(nk), onde k é o número de dígitos.

Bucket Sort:

Divide a lista em ”buckets”, ordena cada ”bucket”e concatena. Eficiente para distribuições uniformes de dados. Complexidade: O(n^2)$nopiorcaso, maslinearemcenáriosespecíficos$.

2. Aleatório

Esta tabela apresenta o desempenho de diferentes algoritmos de ordenação ao lidar com listas geradas aleatoriamente. Os tempos de execução e as comparações serão destacados para proporcionar uma visão abrangente da eficiência de cada algoritmo em cenários diversos.

Aleatório															
	Tempo				Troca				Comparação						
	100	1000	10000		100	1000	10000		100	100	10000				
BubbleSort	0,02	0,15	2,16		4950	258124	25343932		2481	499500	4995000				
Inserção Direta	0,02	0,14	1,61		2580	259123	25353931		2481	258124	25343932				
Inserção Binaria	0,02	0,14	1,59		548	258130	25344544		2481	8756	118942				
Shellsort	0,01	0,14	1,8		938	16305	275452		435	8299	155447				
Selection Sort	0,02	0,13	1,74		97	995	9993		4950	499500	49995000				
Heapsort	0,02	0,14	1,58		575	9075	124717		1029	16845	235337				
Quicksort	0,28	4,06	77,16		?	?	?		?	?	?				
Mergesort	0,64	6,74	120,67		100	1000	10000		99	999	9999				
Radixsort	0,02	0,16	1,78		500	5000	50000		Não faz	Não faz	Não faz				
Bucketsort	0,01	0,15	1,54		100	1000	10000		10	10	10				

Figura 1. Aleatório

3. Crescente

Esta tabela examina o desempenho dos algoritmos de ordenação ao lidar com listas já ordenadas de forma crescente. O foco estará nos tempos de execução e nas comparações, destacando quais algoritmos se destacam em cenários onde a ordem prévia é conhecida.

Crescente														
	Tempo				Troca				Comparação					
	100	1000	10000		100	1000	10000		100	100	10000			
BubbleSort	0,02	0,2	1,85		0	0	0		4950	499500	4995000			
Inserção Direta	0,01	0,14	1,57		0	999	9999		99	0	0			
Inserção Binaria	0,02	0,15	1,74		0	1	317		573	8977	123609			
Shellsort	0,01	0,15	1,62		503	8006	120005		0	0	0			
Selection Sort	0,02	0,14	2,11		0	0	0		4950	499500	49995000			
Heapsort	0,02	0,15	1,48		640	9691	131522		1081	17578	244502			
Quicksort	0,26	6,57	112,44		?	?	?		?	?	?			
Mergesort	0,42	4,06	120,68		100	1000	10000		50	500	5000			
Radixsort	0,02	0,25	1,38		600	6000	60000		Não faz	Não faz	Não faz			
Bucketsort	0,02	1,14	1,7		100	1000	1000		10	10	10			

Figura 2. Crescente

4. Decrescente

Esta tabela explora o desempenho dos algoritmos diante de listas ordenadas de forma decrescente. Os tempos de execução e o número de comparações são destacados para identificar os algoritmos mais eficientes ou adaptáveis em cenários desafiadores.

Descrecente														
	Tempo				Troca				Comparação					
	100	1000	10000		100	1000	10000		100	100	10000			
BubbleSort	0,03	0,15	1,79		4950	499496	49994626		4950	499500	49995000			
Inserção Direta	0,02	0,014	1,99		5049	500495	50004625		4950	499496	49994626			
Inserção Binaria	0,02	0,14	1,68		516	499497	49994841		4950	8472	117572			
Shellsort	0,02	0,17	1,66		763	12704	182418		260	4698	62413			
Selection Sort	0,01	0,15	1,71		50	502	5161		4950	499500	49995000			
Heapsort	0,02	0,14	1,62		516	8318	116688		944	15965	226658			
Quicksort	0,5	6,88	124,21		?	?	?		?	?	?			
Mergesort	0,57	7,26	130,84		100	1000	10000		50	500	5000			
Radixsort	0,02	0,15	1,66		600	6000	60000		Não faz	Não faz	Não faz			
Bucketsort	0,02	0,14	1,77		100	1000	10000		10	10	10			

Figura 3. Decrescente

5. Conclusão

O desempenho de algoritmos de ordenação é crucial para eficiência computacional. Em listas aleatórias, Quick Sort e Merge Sort se destacam, enquanto Bubble Sort e Selection Sort podem ser menos eficientes.

Listas quase ordenadas beneficiam-se de métodos de inserção, como Bubble Sort e Insertion Sort. Em listas crescentes, algoritmos baseados em comparações, como Quick Sort, são eficazes, enquanto em listas decrescentes, Heap Sort e Merge Sort mostram vantagens.

Além da análise teórica, a implementação específica e otimizações influenciam o desempenho real. Testes empíricos são essenciais para a escolha apropriada do algoritmo, garantindo eficiência em diversos contextos.

6. Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.

Sedgewick, R., Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

Dasgupta, S., Papadimitriou, C. H., Vazirani, U. V. (2008). Algorithms. McGraw-Hill.

Sedgewick, R. (1998). Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching (3rd ed.). Addison-Wesley.