Radixsort

Adriel Jhames, Leonardo Sade, Thiago Galego

Introdução

O Radixsort é um algoritmo de ordenação rápida e estável, no qual é usado para ordenar strings e chaves numéricas, por meio de baldes, no qual vai percorrer dígito por dígito do menos significante para o mais significante, decrementando assim e ordenando de trás para frente, de acordo com Cormen et al. [1] O Radix não foi uma apenas uma ideia teórica, ou seja, ele surgiu da prática física de ordenar dados, como antes dos computadores digitais, os dados eram armazenados em cartões perfudaros, no qual as maquinas de ordenação utilizavam um principio semelhante ao Radix Sorrt para ordenar grandes volumes de dados.

Como cada cartão perfurado tinha 80 colunas para os caracteres, e cada coluna podia representar um caractere usando 12 possíveis furos, sendo 10 para os dígitos de 0 a 9 e mais 2 para letras e símbolos especiais. A ideia de programar mecanicamente significa ajustar a máquina para olhar, por exemplo, a coluna 5 de todos os cartões e colocá-los em diferentes caixas conforme o valor perfurado.

Isto é o passo de ordenação estável do RadixSort no qual os cartões eram recolhidos na ordem dos compartimentos, preservando a ordenação anterior. Esse processo repetia-se de coluna para coluna, do dígito menos significativo para o mais significativo, exatamente como o RadixSort ordena.

Sua grande vantagem é no qual ordena em tempo linear O(N) para arrays positivos e não tão longos, já para arrays maiores e com mais números significativos sua complexidade de tempo aumenta para O(D + N).

Algoritmo

O Radixsort é um algoritmo de ordenação no qual classifica elementos e os processa dígito por dígito do menos significativo para o mais significativo, e para ordenação depende muito de sua base, sendo assim podendo ser Base 10 no qual consiste os números inteiros. Para a ordenação, o Radix primeiramente pega o dígito menos significativo. Usaremos como exemplo o array da Tabela 1.

De forma matemática, o Radixsort utiliza-se do exponencial para então deduzir o número menos significativo, sendo assim ele irá iniciar no expoente 10^0 e assim sucessivamente. Por exemplo, o número 459, sabemos que o 4 é a centena dada a fórmula matemática 4.10^2, o 5 a dezena 4.10^1 e o 9 a unidade 9. 10^0. Sendo assim, começamos pelo dígito das unidades, da esquerda para a direita, e então começa a ordenação por balde, onde vai separar de 0 a 9 os baldes e juntar todos que terminam com seus respectivos números, assim como na tabela 2.

170	**45**	**75**	**90**	**802**	**24**	**2**	**66**	**459**

Table 1: Vetor original de entrada para o Radix Sort

Balde (Dígito)	**Números**
0	170, 90
1	
2	802, 2
3	
4	24
5	45, 75
6	66
7	
8	
9	459

Table 2: Distribuição dos números pelo dígito das unidades

Após esta separação, ele começa a ordenação da esquerda para a direita, usando o vetor principal, a posição zero encontra o número, no qual vai observar o número menos significativo e então separando-o conforme a tabela 2 e colocando os números em ordem, assim ficando igual à tabela 3. Note que o 170 continua na frente, isto é pelo fato de o algoritmo ser estável, ele irá manter o número antes, agora o processo irá se repetir novamente para o próximo número menos significante que seria a casa das dezenas, observe a próxima tabela a seguir:

Balde (Dígito)	Números
0	802, 2
1	
2	24
3	
4	45
5	459
6	66
7	170, 75
8	
9	90

Table 3: Distribuição dos números pelo dígito das dezenas

Novamente agora aproveitando o mesmo array vai ordenar, mas desta vez utlizando a casa decimal como base sendo assim atualizaremos nossa ordem para: 802, 02, 24, 459, 66, 170, 75, 90 e agora por fim o ultimo digito restante

Balde (Dígito)	**Números**
0	45, 75, 90, 24, 2, 66
1	170
2	
3	
4	459
5	
6	
7	
8	802
9	

Table 4: Distribuição dos números pelo dígito das centenas

E por fim temos a nossa ordenação realizada e assim obtendo a ordenação certa do array para 2, 24, 45, 66, 75, 90, 170, 459, 802.

De acordo com Weiss [8], a ordenação por baldes torna o algorítimo em casos especiais para a comparação de tempo linear. Sendo assim, a entrada A1, A2, ..., AN deve consistir em apenas inteiros positivos menores que M. Sendo assim, conforme o mesmo, o array vai chamar o count, de tamanho M, ou baldes que estão vazios. Quando Ai for lido e incrementado de count[Ai] em 1, após isso, toda entrada lida vai examinar o array count, imprimindo uma representação ordenada. Assim, o Algoritmo recebe O(M + N). Se M for O(N), então o total é um O(N). Ao adicionar o balde apropriado, o algoritmo realiza uma comparação M em tempo unitária, sendo semelhante a estratégia hashing extensível.

Implementação e Testes

Para fins comparativos, realizamos a implementação e o teste com um array de tamanho 4 milhões, no qual utilizamos um array ordenado, reverso e uniforme para comparar o Radixsort com o SelectionSort, InsertionSort e QuickSort.

Gorset [2] afirma que para matrizes de inteiros, a classificação por radix supera a classificação rápida em teoria, velocidade real, clareza algorítmica e legibilidade. Portanto, primeiro iremos apresentar nossos testes referentes ao Quicksort.

Table 5: Tempo de execução do QuickSort (em segundos) para diferentes tamanhos de vetor

Vector Size	**Ordered_Array**	**Reverse_Array**	**Uniform_Array**
1.000	0,000	0,000	0,001
50.000	0,002	0,002	0,008
100.000	0,004	0,004	0,018
200.000	0,008	0,009	0,037
300.000	0,013	0,015	0,058
400.000	0,017	0,019	0,080
500.000	0,023	0,025	0,100
600.000	0,028	0,030	0,121
700.000	0,033	0,039	0,145
800.000	0,036	0,039	0,165
900.000	0,042	0,044	0,189
1.000.000	0,054	0,050	0,210
2.000.000	0,099	0,111	0,440
3.000.000	0,154	0,161	0,679
4.000.000	0,208	0,223	0,921

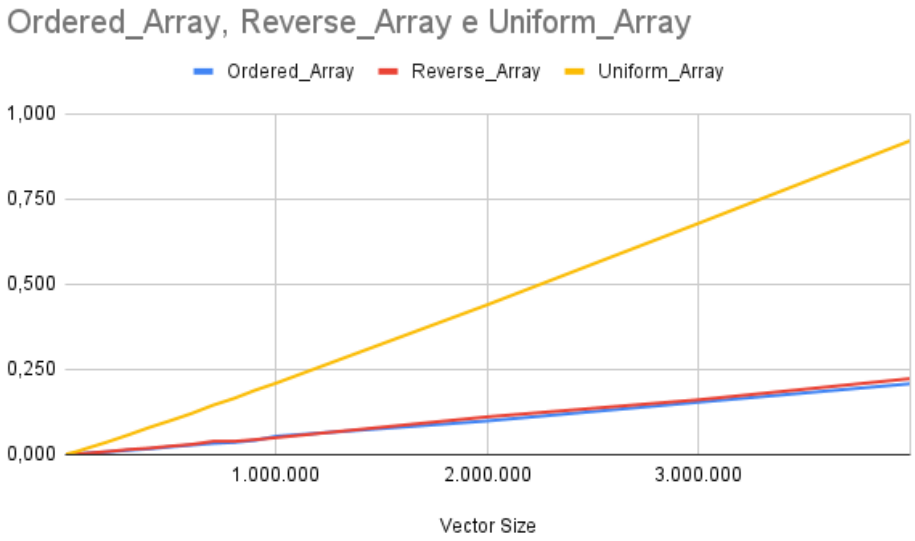

Figure 1: QuickSort Gráfico Comparativo.

Table 6: Tempo de execução do Radix Sort (em segundos) para diferentes tamanhos de vetor

Vector Size	**Ordered_Array**	**Reverse_Array**	**Uniform_Array**
1.000	0,000	0,000	0,000
50.000	0,000	0,016	0,010
100.000	0,000	0,000	0,000
200.000	0,000	0,000	0,010
300.000	0,000	0,015	0,016
400.000	0,015	0,016	0,015
500.000	0,016	0,031	0,016
600.000	0,031	0,031	0,026
700.000	0,032	0,031	0,026
800.000	0,031	0,031	0,031
900.000	0,031	0,047	0,036
1.000.000	0,046	0,047	0,047
2.000.000	0,094	0,094	0,083
3.000.000	0,156	0,140	0,135
4.000.000	0,188	0,203	0,177

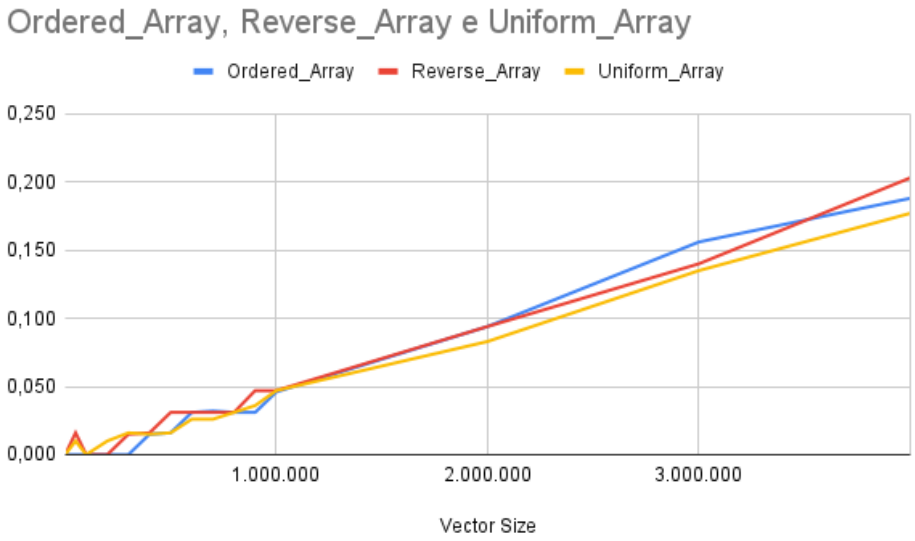

Figure 2: RadixSort Gráfico Comparativo.

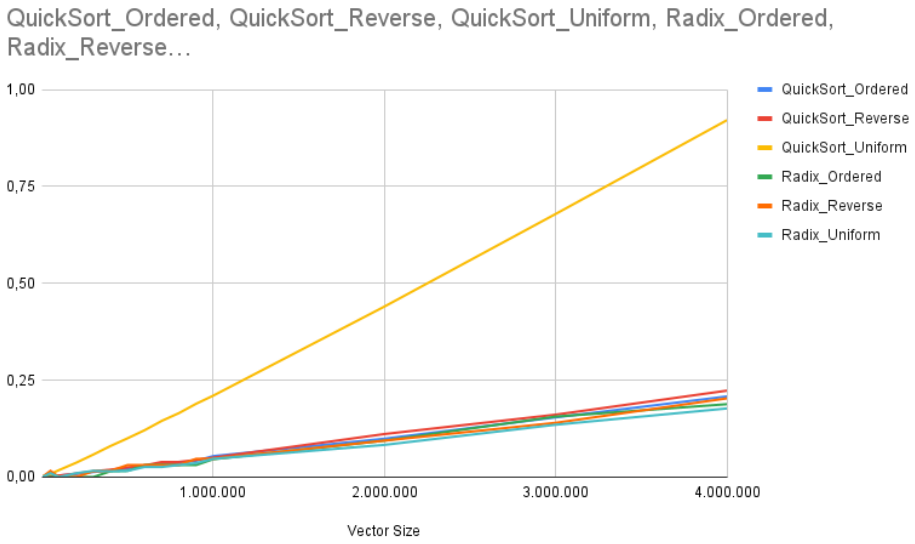

Figure 3: QuickSort Gráfico Comparativo.

Para Sedgewick e Wayne [7] a complexidade do RadixSort é de O(W.N), onde o W é o número de caracteres ou dígitos e o N é o número de itens a ordenar, sendo assim afirma-se que para strings fixas de comprimento W e alfabeto de tamanho R, a complexidade de RadixSort é linear em relação ao tamanho da entrada.

Na figura 3 podemos ver a comparação de tempo entre os mesmos, no qual afirmamos

que para números inteiros o RadixSort consegue ser mais rápido do que o Quicksort.

Conclusão

Com base nos testes realizados e na análise teórica, foi possível observar que o Radixsort se destaca como uma alternativa eficiente para ordenação de grandes volumes de dados inteiros. Sua abordagem, baseada na ordenação dígito a dígito, permite alcançar tempos de execução inferiores aos de algoritmos tradicionais em determinados cenários, especialmente quando os dados estão em formatos uniformes ou parcialmente ordenados.

Além disso, sua estabilidade e capacidade de manter a ordem relativa dos elementos tornam o algoritmo ainda mais vantajoso em aplicações específicas. A implementação prática comprovou que, para arrays com milhões de elementos, o Radixsort apresenta ganhos consideráveis de desempenho.

Dessa forma, concluímos que o Radixsort é uma solução eficaz e competitiva dentre os algoritmos de ordenação, principalmente quando se trata de dados inteiros e cenários onde o desempenho e a estabilidade são fatores decisivos.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. Capítulo 8: Sorting in Linear Time.

[2] Erik Gorset. Radix sort is faster than quicksort, April 2011. Acessado em 9 de junho de 2025.

[3] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. Seção 5.2.5 – Digital Sorting.

[4] Daniel Lemire. I wrote a faster sorting algorithm, December 2016. Acessado em 9 de junho de 2025.

[5] David M. W. Powers. Parallelized quicksort and radixsort with optimal speedup. In *Proceedings of International Conference on Parallel Computing Technologies*, Novosibirsk, November 1991.

[6] Nadathur Satish, Mark Harris, and Michael Garland. Engineering radix sort for gpus. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 246–256, 2009. Acessado em 9 de junho de 2025.

[7] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011. Seção 5.1: Radix Sorting.

[8] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Pearson, 4th edition, 2013. Capítulo 7 - Sorting; Capítulo: Non-Comparison-Based Sorting; Capítulo: Sorting Algorithms.