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, ou seja, supondo que há um número com 3 dígitos, começará pelas unidades, assim ordenando-as, após isso vai para as dezenas também as ordenando e retornando o array ordenado e por fim as centenas, no qual vai deixar tudo ordenado corretamente.

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 e Base 256 utilizando a tabela ASCII onde possui um valor número de 0 a 255 sendo essas duas as mais utilizadas, já em cenários mais específicos utilizam-se também da Base 16 (Hexadecimal) no qual é uma representação mais compacta dos números e a Base 2 que é o sistema binário

no qual é usado pelos computadores. 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 10^1 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	
9	802

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. Conforme Satish et al. [6], conjuntos de baldes podem ser gerenciados com uma simples estrutura de array de ponteiros, que é efetivamente um nó trie. 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.

```
Sorting files with format: input/ordered/ordered-input-%d-float.txt
Quicksort [01/15] - Vector Size: 10000 - CPU time : 0.000000 seconds
Quicksort [02/15] - Vector Size: 50000 - CPU time : 0.002000 seconds
Quicksort [03/15] - Vector Size: 100000 - CPU time : 0.004000 seconds
Quicksort [04/15] - Vector Size: 200000 - CPU time : 0.008000 seconds
Quicksort [05/15] - Vector Size: 300000 - CPU time : 0.013000 seconds
Quicksort [06/15] - Vector Size: 400000 - CPU time : 0.017000 seconds
Quicksort [07/15] - Vector Size: 500000 - CPU time : 0.023000 seconds
Quicksort [08/15] - Vector Size: 600000 - CPU time : 0.028000 seconds
Quicksort [09/15] - Vector Size: 700000 - CPU time : 0.033000 seconds
Quicksort [10/15] - Vector Size: 800000 - CPU time : 0.036000 seconds
Quicksort [11/15] - Vector Size: 900000 - CPU time : 0.042000 seconds
Quicksort [12/15] - Vector Size: 1000000 - CPU time : 0.054000 seconds
Quicksort [13/15] - Vector Size: 2000000 - CPU time : 0.099000 seconds
Quicksort [14/15] - Vector Size: 3000000 - CPU time : 0.154000 seconds
Quicksort [15/15] - Vector Size: 4000000 - CPU time : 0.208000 seconds
```

Figure 1: QuickSort Ordenado.

```
Sorting files with format: input/reverse_ordered/reverse_ordered-input-%d-float.txt
Quicksort [01/15] - Vector Size: 10000 - CPU time : 0.000000 seconds
Quicksort [02/15] - Vector Size: 50000 - CPU time : 0.002000 seconds
Quicksort [03/15] - Vector Size: 100000 - CPU time : 0.004000 seconds
Quicksort [04/15] - Vector Size: 200000 - CPU time : 0.009000 seconds
Quicksort [05/15] - Vector Size: 300000 - CPU time : 0.015000 seconds
Quicksort [06/15] - Vector Size: 400000 - CPU time : 0.019000 seconds
Quicksort [07/15] - Vector Size: 500000 - CPU time : 0.025000 seconds
Quicksort [08/15] - Vector Size: 600000 - CPU time : 0.030000 seconds
Quicksort [09/15] - Vector Size: 700000 - CPU time : 0.039000 seconds
Quicksort [10/15] - Vector Size: 800000 - CPU time : 0.039000 seconds
Quicksort [11/15] - Vector Size: 900000 - CPU time : 0.044000 seconds
Quicksort [12/15] - Vector Size: 1000000 - CPU time : 0.050000 seconds
Quicksort [13/15] - Vector Size: 2000000 - CPU time : 0.111000 seconds
Quicksort [14/15] - Vector Size: 3000000 - CPU time : 0.161000 seconds
Quicksort [15/15] - Vector Size: 4000000 - CPU time : 0.223000 seconds
```

Figure 2: QuickSort ordem Reversa.

```
Sorting files with format: input/uniform/uniform-input-%d-%d-float.txt
Quicksort [01/15] - Vector Size: 10000 - CPU time : 0.001333 seconds
Quicksort [02/15] - Vector Size: 50000 - CPU time : 0.008333 seconds
Quicksort [03/15] - Vector Size: 100000 - CPU time : 0.018000 seconds
Quicksort [04/15] - Vector Size: 200000 - CPU time : 0.037000 seconds
Quicksort [05/15] - Vector Size: 300000 - CPU time : 0.058333 seconds
Quicksort [06/15] - Vector Size: 400000 - CPU time : 0.080000 seconds
Quicksort [07/15] - Vector Size: 500000 - CPU time : 0.100667 seconds
Quicksort [08/15] - Vector Size: 600000 - CPU time : 0.121333 seconds
Quicksort [09/15] - Vector Size: 700000 - CPU time : 0.145000 seconds
Quicksort [10/15] - Vector Size: 800000 - CPU time : 0.165667 seconds
Quicksort [11/15] - Vector Size: 900000 - CPU time : 0.189333 seconds
Quicksort [12/15] - Vector Size: 1000000 - CPU time : 0.210000 seconds
Quicksort [13/15] - Vector Size: 2000000 - CPU time : 0.440333 seconds
Quicksort [14/15] - Vector Size: 3000000 - CPU time : 0.679333 seconds
Quicksort [15/15] - Vector Size: 4000000 - CPU time : 0.921000 seconds
```

Figure 3: QuickSort ordem Uniforme.

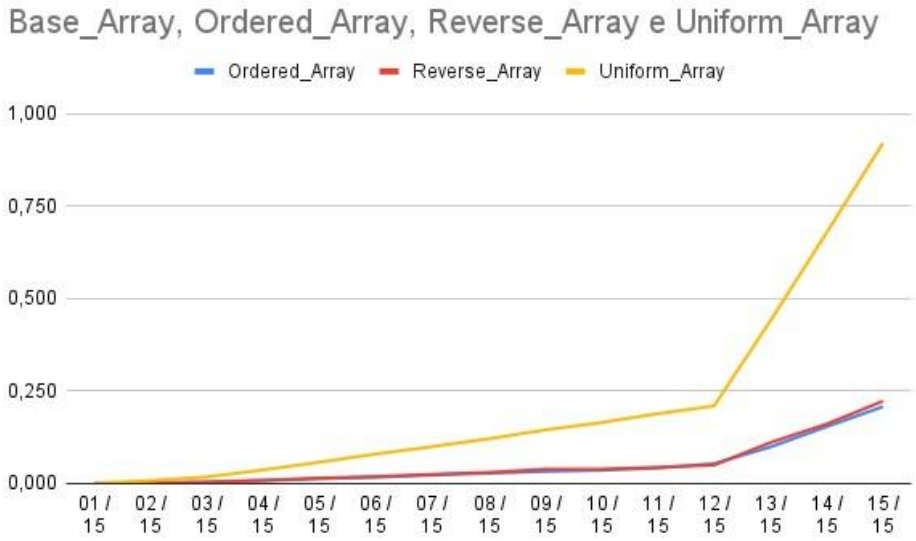

Figure 4: QuickSort Gráfico Comparativo.

Na figura 4 podemos acompanhar o tempo entre os Arrays Ordenados, Reverso e Uniforme.

```
Sorting files with format: input/ordered/ordered-input-%d-float.txt
Radix Sort [01/15] - Vector Size: 10000 - CPU time : 0.000000 seconds
Radix Sort [02/15] - Vector Size: 50000 - CPU time : 0.000000 seconds
Radix Sort [03/15] - Vector Size: 100000 - CPU time : 0.000000 seconds
Radix Sort [04/15] - Vector Size: 200000 - CPU time : 0.000000 seconds
Radix Sort [05/15] - Vector Size: 300000 - CPU time : 0.016000 seconds
Radix Sort [06/15] - Vector Size: 400000 - CPU time : 0.015000 seconds
Radix Sort [07/15] - Vector Size: 500000 - CPU time : 0.016000 seconds
Radix Sort [08/15] - Vector Size: 600000 - CPU time : 0.031000 seconds
Radix Sort [09/15] - Vector Size: 700000 - CPU time : 0.032000 seconds
Radix Sort [10/15] - Vector Size: 800000 - CPU time : 0.031000 seconds
Radix Sort [11/15] - Vector Size: 900000 - CPU time : 0.031000 seconds
Radix Sort [12/15] - Vector Size: 1000000 - CPU time : 0.046000 seconds
Radix Sort [13/15] - Vector Size: 2000000 - CPU time : 0.094000 seconds
Radix Sort [14/15] - Vector Size: 3000000 - CPU time : 0.156000 seconds
Radix Sort [15/15] - Vector Size: 4000000 - CPU time : 0.188000 seconds
```

Figure 5: RadixSort Ordenado.

```
Sorting files with format: input/reverse_ordered/reverse_ordered-input-%d-float.txt
Radix Sort [01/15] - Vector Size: 10000 - CPU time : 0.000000 seconds
Radix Sort [02/15] - Vector Size: 50000 - CPU time : 0.016000 seconds
Radix Sort [03/15] - Vector Size: 100000 - CPU time : 0.000000 seconds
Radix Sort [04/15] - Vector Size: 200000 - CPU time : 0.000000 seconds
Radix Sort [05/15] - Vector Size: 300000 - CPU time : 0.015000 seconds
Radix Sort [06/15] - Vector Size: 400000 - CPU time : 0.016000 seconds
Radix Sort [07/15] - Vector Size: 500000 - CPU time : 0.031000 seconds
Radix Sort [08/15] - Vector Size: 600000 - CPU time : 0.031000 seconds
Radix Sort [09/15] - Vector Size: 700000 - CPU time : 0.031000 seconds
Radix Sort [10/15] - Vector Size: 800000 - CPU time : 0.031000 seconds
Radix Sort [11/15] - Vector Size: 900000 - CPU time : 0.047000 seconds
Radix Sort [12/15] - Vector Size: 1000000 - CPU time : 0.047000 seconds
Radix Sort [13/15] - Vector Size: 2000000 - CPU time : 0.094000 seconds
Radix Sort [14/15] - Vector Size: 3000000 - CPU time : 0.140000 seconds
Radix Sort [15/15] - Vector Size: 4000000 - CPU time : 0.203000 seconds
```

Figure 6: RadixSort ordem Reversa.

```
Sorting files with format: input/uniform/uniform-input-%d-%d-float.txt
Radix Sort [01/15] - Vector Size: 10000 - CPU time : 0.000000 seconds
Radix Sort [02/15] - Vector Size: 50000 - CPU time : 0.010333 seconds
Radix Sort [03/15] - Vector Size: 100000 - CPU time : 0.000000 seconds
Radix Sort [04/15] - Vector Size: 200000 - CPU time : 0.010667 seconds
Radix Sort [05/15] - Vector Size: 300000 - CPU time : 0.016000 seconds
Radix Sort [06/15] - Vector Size: 400000 - CPU time : 0.015667 seconds
Radix Sort [07/15] - Vector Size: 500000 - CPU time : 0.016000 seconds
Radix Sort [08/15] - Vector Size: 600000 - CPU time : 0.026333 seconds
Radix Sort [09/15] - Vector Size: 700000 - CPU time : 0.026000 seconds
Radix Sort [10/15] - Vector Size: 800000 - CPU time : 0.031333 seconds
Radix Sort [11/15] - Vector Size: 900000 - CPU time : 0.036333 seconds
Radix Sort [12/15] - Vector Size: 1000000 - CPU time : 0.047000 seconds
Radix Sort [13/15] - Vector Size: 2000000 - CPU time : 0.083333 seconds
Radix Sort [14/15] - Vector Size: 3000000 - CPU time : 0.135000 seconds
Radix Sort [15/15] - Vector Size: 4000000 - CPU time : 0.177333 seconds
```

Figure 7: RadixSort ordem Uniforme.

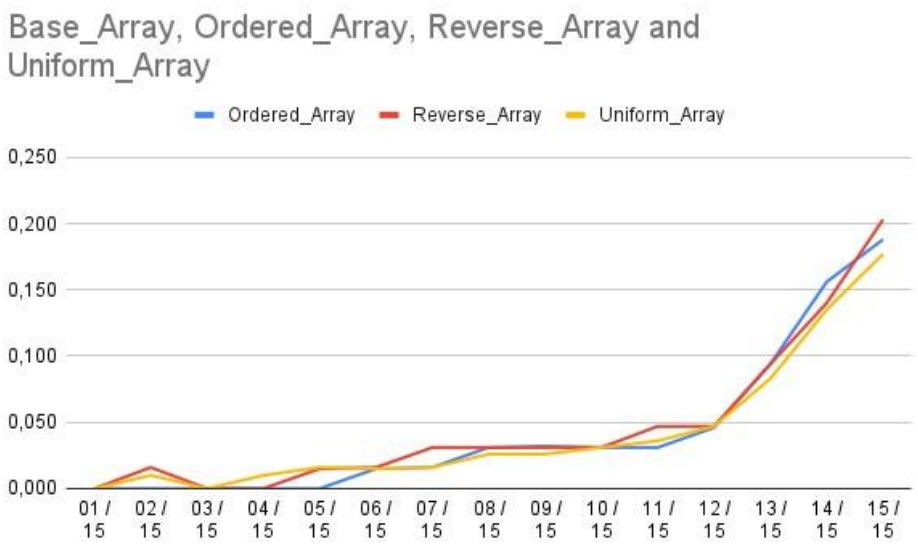

Figure 8: QuickSort Gráfico Comparativo.

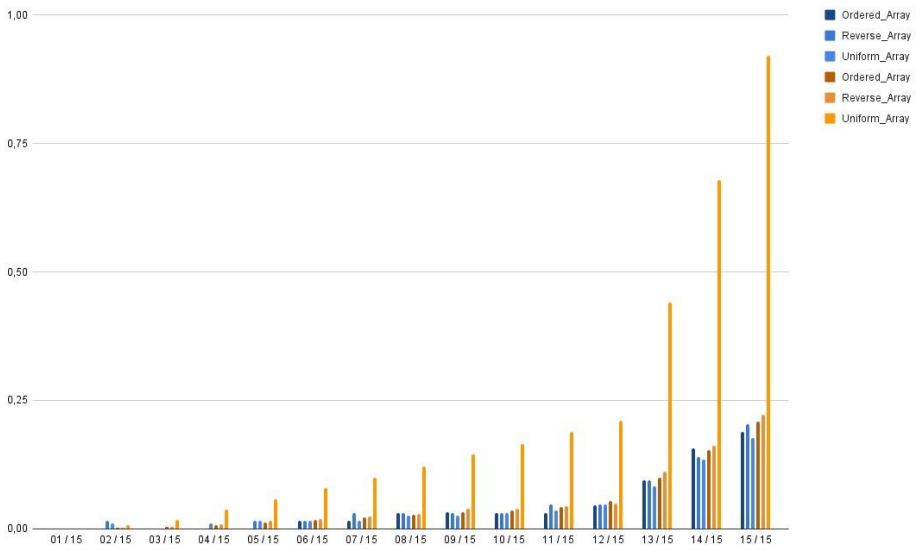

Figure 9: Comparativo entre QuickSort e RadixSort.

Na figura 9 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.