Algoritmos de Ordenação da Disciplina de Estrutura de Dados II

Carlos Eduardo Rocha Miranda

[1]Instituto Federal Goiano - Campus Morrinhos(IF Goiano)
Rodovia BR153 - KM633 Zona Rural - Morrinhos - GO - 75650-000

{eduardo,carlos}carlos.eduardo@estudante.ifgoiano.edu.br

Abstract. *This article deals with all sorting algorithms studied in the Data Structure II course at IF Goiano - Campus Morrinhos. It explains the operation of the algorithm, its number of comparisons, its number of movements and its execution time.*

Resumo. *Este artigo trata de todos os algoritmos de ordenação estudados na disciplina de Estrutura de Dados II do IF Goiano - Campus Morrinhos. Nele está explicitado o funcionamento do algoritmo, a sua quantidade de comparações, a sua quantidade de movimentações e o seu tempo de execução.*

1. Insertion Sort

É um algoritmo de ordenação que divide os números em duas áreas. A área dos ordenados e a área dos não-ordenados.No começo a área dos ordenados possui apenas um valor, portanto está ordenada. Logo em seguida se pega o primeiro valor da área não ordenada e o comparamos com os valores presentes na área ordenada, se é maior ou menor. A depender do resultado o valor é posicionado dentro da área dos ordenados. Confira a figura 1.

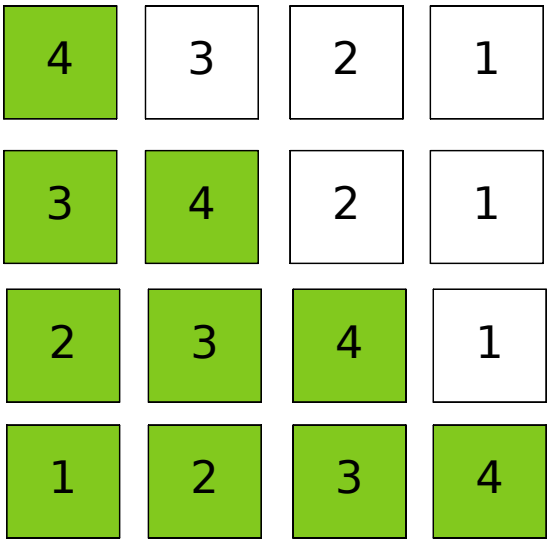

Figura 1. Insertion Sort

Sua implementação ocorreu sem nenhum tipo de problema. A consulta dos materiais disponibilizados pelo professor e também outros avulsos facilitaram o entendimento

Tabela 1. Insertion Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	13	9
100	00:00:00:00	2639	2540
1.000	00:00:00:00	251978	250979
10.000	00:00:00:01	25299924	25289925
50.000	00:00:01:05	625115591	625065592
100.000 (caso médio)	00:00:07:03	2504253129	2504153130
100.000 (pior caso)	00:00:12:07	4999983755	4999883756
100.000 (melhor caso)	00:00:00:00	199998	99999
500.000	00:11:47:04	62507458280	62506958281

do algoritmo. Ele possui: no pior caso $O(n^2)$, no caso médio $O(n^2)$ e no melhor caso $O(n)$. Confira sua tabela de dados 1.

2. Selection Sort

Esse algoritmo é um pouco diferente do Insertion Sort e imita a maneira como uma criança ordena as coisas. Existe um laço de repetição que passa através dos números procurando o menor número possível. Ao encontrá-lo dentre todos, o reposiciona na primeira posição. Logo em seguida, a partir da segunda posição, ele procura o menor valor posição dentro da área dos não ordenadas e assim por diante. Confira a figura 2.

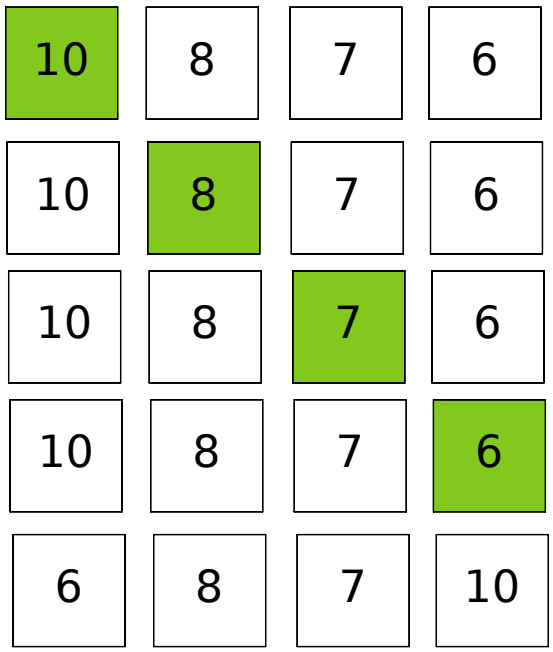

Figura 2. Selection Sort

Sua implementação, semelhante ao Insertion Sort, ocorreu sem maiores problemas. Na realidade, a forma como o algoritmo foi pensado é extremamente natural e sua complexidade seguiu sendo verdadeira em sua tabela. Ele possui: no pior caso $O(n^2)$, no caso médio $O(n^2)$ e no melhor caso $O(n^2)$. Confira sua tabela de dados 2.

Tabela 2. Selection Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	12	29
100	00:00:00:00	297	10099
1.000	00:00:00:00	2997	1000999
10.000	00:00:00:01	29997	100009999
50.000	00:00:02:00	149997	2500049999
100.000 (caso médio)	00:00:08:05	299997	10000099999
100.000 (pior caso)	00:00:13:06	299997	10000099999
100.000 (melhor caso)	00:00:05:06	199998	10000099999
500.000	00:05:46:06	1499997	250000499999

3. Bubble Sort

É um algoritmo de ordenação que vai trocando de lugar os elementos adjacentes, sempre direcionando o maior valor possível para a última posição do array. Ele vai comparando cada valor procurando o maior deles. Ao achar o maior ele troca de lugar, avançando uma posição no array. Isso vai acontecer até o maior valor possível encontrar a última posição e nas repetições seguintes, o segundo maior valor, o terceiro maior valor, etc. Confira a figura 3:

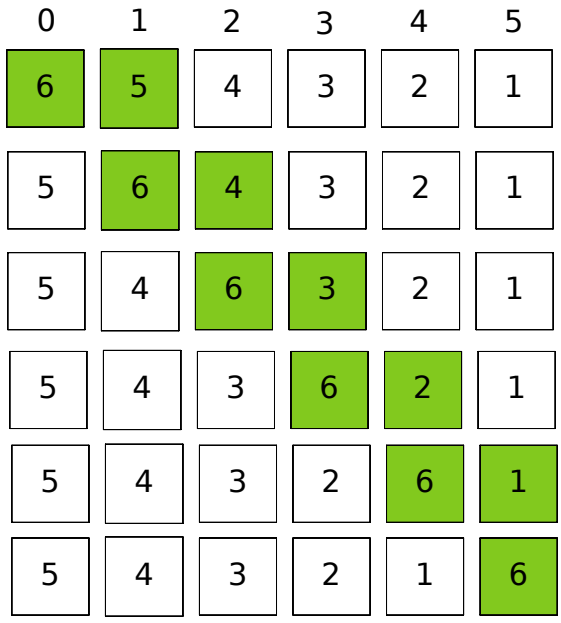

Figura 3. Bubble Sort

Sua implementação é divertida. O ato de subir os valores mais altos para o final do array é extremamente interessante. Outra parte curiosa desse algoritmo é o fato de não haver trocas quando o array já está previamente ordenado. Ele possui: no pior caso $O(n^2)$, no caso médio $O(n^2)$ e no melhor caso $O(n)$. Confira sua tabela de dados 3.

Tabela 3. Bubble Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	15	45
100	00:00:00:00	7323	19900
1.000	00:00:00:00	749940	1999000
10.000	00:00:00:07	75839778	199990000
50.000	00:00:23:09	1875046779	4999950000
100.000 (caso médio)	00:01:42:08	7512159393	19999900000
100.000 (pior caso)	00:02:07:02	14999351271	19999900000
100.000 (melhor caso)	00:00:05:06	0	19999900000
500.000	00:53:55:06	187519374846	499999500000

4. Combo Sort

É um algoritmo de ordenação que é uma espécie de Bubble Sort aprimorado. A ideia dele é fazer as trocas que o Bubble Sort faz de maneira espaçada, usando um passo de $h/1.3$. Onde h, no começo, é o tamanho do array. A cada repetição é refeita a conta do $h/1.3$ e portanto as espaçadas diminuem de tamanho até atingirem o valor de 1, onde ocorre, obviamente, um bubble sort comum para terminar de ordenar as partes que ainda não estão ordenados. Confira a figura 4:

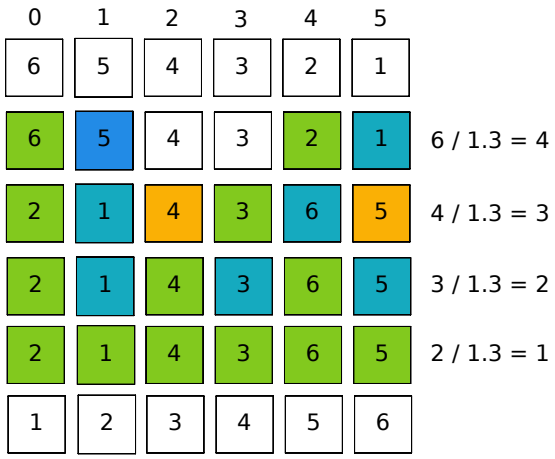

Figura 4. Combo Sort

Sua implementação foi um pouco mais complexa. Como ele utiliza uma estratégia de divisão e conquista, as coisas nesse algoritmo se tornam mais abstratas. Ele, como o Bubble Sort, não realiza trocas em um array que já está ordenado. Ele possui: no pior caso $O(n^2)$, no caso médio $O((n^2)/2^p)$ e no melhor caso $O(nlogn)$. Confira sua tabela de dados 4.

5. Shell Sort

É um algoritmo de ordenação que funciona ordenando partes subdivididas graças a uma propriedade conhecida como fator de encolhimento. A partir do meu fator de encolhimento(que é calculado pela fórmula $(h * 3 + 1)/3$) o array é dividido em partes e nessas

Tabela 4. Combo Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	9	30
100	00:00:00:00	777	2417
1.000	00:00:00:00	13266	43445
10.000	00:00:00:00	188601	653504
50.000	00:00:00:00	1121172	4366851
100.000 (caso médio)	00:00:00:00	2218323	8133526
100.000 (pior caso)	00:00:00:00	669249	7533529
100.000 (melhor caso)	00:00:00:00	0	7333530
500.000	00:00:00:04	11665287	48666894

partes específicas o algoritmo Insertion Sort é aplicado. O fator de encolhimento diminui a cada passada(como o próprio nome sugere) e a cada passada o array é subdividido mais uma vez e o algoritmo Insertion Sort é aplicado mais uma vez até que todo o array esteja ordenado. Confira a figura 5:

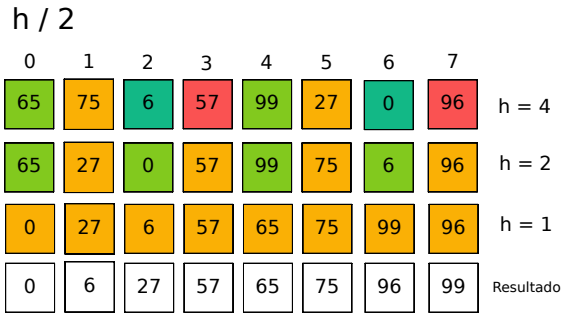

Figura 5. Shell Sort

Sua implementação ocorreu sem maiores problemas, mas a sua compreensão foi difícil. É um algoritmo que subdivide suas partes a partir de uma fator de encolhimento e nessas partes subdivididas ele aplica outro algoritmo de ordenação para ordenar o array. E é exatamente por isso que foi complicado compreendê-lo e explicá-lo nos vídeos. Ele possui: no pior caso $O(nlog_2n)$, no caso médio sua complexidade depende do gap e no melhor caso $O(nlog_2n)$. Confira sua tabela de dados 5.

6. Bogo Sort

É um algoritmo conhecido por sua estupidez. Ele consiste completamente na sorte. Você pega o array, verifica se está ordenado, caso não esteja você o embaralha e torce pelo melhor. Ele é totalmente ruim e depende do acaso para o seu bom funcionamento. Confira a figura 6.

Sua implementação, dentre todos até agora, foi a mais trabalhosa. Isso é contra intuitivo, mas o problema era que o método de embaralhamento utilizado havia sido implementado de modo errado, por conta disso o algoritmo nunca funcionava. Ele possui: no pior caso $O((n+1)!)$, no caso médio $O((n+1)!)$ e no melhor caso $O(n)$. Confira sua tabela de dados 6.

Tabela 5. Shell Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	11	27
100	00:00:00:00	921	1459
1.000	00:00:00:00	16326	24627
10.000	00:00:00:00	261466	381828
50.000	00:00:00:00	1762403	2453403
100.000 (caso médio)	00:00:00:00	4153791	5676702
100.000 (pior caso)	00:00:00:00	2005882	3528793
100.000 (melhor caso)	00:00:00:00	1522866	3045777
500.000	00:00:00:06	27340478	36309097

Tabela 6. Bogo Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	1350	966
100	-	-	-
1.000	-	-	-
10.000	-	-	-
50.000	-	-	-
100.000 (caso médio)	-	-	-
100.000 (pior caso)	-	-	-
100.000 (melhor caso)	-	-	-
500.000	-	-	-

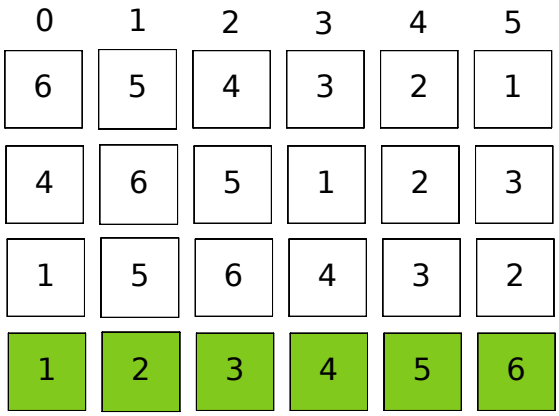

Figura 6. Bogo Sort

7. Quick Sort

É um algoritmo de ordenação que observou algo simples: Num array ordenado, pegando qualquer número aleatoriamente, todos os valores quee stão atrás do número são menores do que ele e todos os valores que estão na frente do número são maiores que ele. E é exatamente isso com o que ele se preocupa. Esse algoritmo escolhe um pivô e se preocupa em apenas colocar os números maiores que o pivõ na frente dele e os números menores que o pivõ atrás dele. Isso é feito recursivamente a exaustão até todo o array estar completamente ordenado. Nos pormenores, temos duas zonas, dos elementos menores que o pivô(representado pela barra azul) e a zona dos elementos maiores(representado pela barra verde) que o pivô. Depois o pivô(que nessa implementação está na última posição) é colocado entre essas duas barras. Note que ele está na sua posição final em todo o array. Logo em seguida as zonas anteriores realizarão o quick sort dentro de si mesmas e assim por diante. Confira a figura 7.

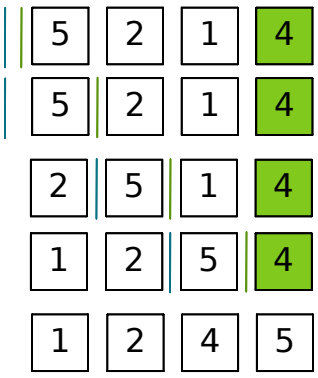

Figura 7. Quick Sort

Sua implementação foi assustadora. O primeiro algoritmo implementado deu completamente errado. Por alguma razão as inúmeras recursões causavam um erro invisível que o Java não mostrava para o usuário. Para consertar isso, foi colocado um try catch em cima das recursões, mas logo então veio um erro na ordenação dos arquivos. Eles estavam parcialmente ordenados e repicados. Para consertar todos esses problemas

Tabela 7. Quick Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	24	20
100	00:00:00:00	777	946
1.000	00:00:00:00	10140	14476
10.000	00:00:00:00	124275	198895
50.000	00:00:00:00	693462	1185042
100.000 (caso médio)	00:00:00:00	1438929	2588880
100.000 (pior caso)	00:00:00:00	659244	2270969
100.000 (melhor caso)	00:00:00:00	512859	2235539
500.000	00:00:00:02	7649553	16940870

a solução foi apagar tudo e refazer o algoritmo do 0. Ele possui: no pior caso $O(n^2)$, no caso médio $O(nlog_2n)$ e no melhor caso $O(nlog_2n)$. Confira sua tabela de dados 7.

8. Merge Sort

É um algoritmo de ordenação totalmente baseado na ideia de dividir para conquistar. Nele, dividimos o array em subarrays até chegarmos em valores unitários. Depois disso começamos a juntar o array na ordem correta até que ele retorne ao seu tamanho original, mas com todos os números corretamente ordenados. Confira a figura 8.

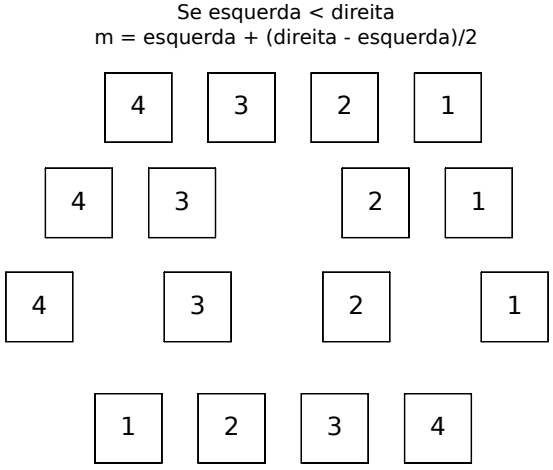

Figura 8. Merge Sort

Sua implementação foi mais difícil que o seu entendimento. A ideia de dividir para conquistar é simples, mas extremamente complexa no caso do Merge Sort. Com sorte, foi possível consultar vídeos explicativos que auxiliaram em sua implementação. Ele possui: no pior caso $O(nlogn)$, no caso médio $O(nlogn)$ e no melhor caso $O(nlogn)$. Confira sua tabela de dados 8.

9. Heap Sort

É um algoritmo de ordenação baseado na ideia de uma árvore do tipo heap. Uma árvore heap é uma árvore binária onde os pais são maiores que os filhos. Ela é criada a partir de

Tabela 8. Merge Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	17	16
100	00:00:00:00	795	748
1.000	00:00:00:00	11232	10719
10.000	00:00:00:00	146641	140590
50.000	00:00:00:00	850793	818134
100.000 (caso médio)	00:00:00:00	1801703	1736152
100.000 (pior caso)	00:00:00:00	2455441	1082414
100.000 (melhor caso)	00:00:00:00	2522832	1015023
500.000	00:00:00:01	10114102	9837321

um array, tomando sua posição inicial, o seu filho esquerdo será igual a 2*i + 1 e o seu filho direito será 2*i + 2. Essa conta é feita sucessivamente em cada posição até chegar na metade do array(depois da metade só haverão filhos nulos). Confira a figura 9.

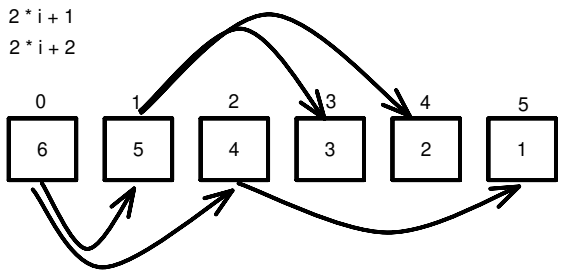

Figura 9. Heap Sort

Depois de montada a estrutura da árvore heap(confira a figura 10), o algoritmo pega a raiz e troca ela de lugar com o menor valor possível na árvore, logo em seguida esse menor valor é removido e recursivamente o heap é mais uma vez montada(pois sua estrutura foi destruída com essa troca). O valor removido é sempre colocado no final do array. A cada vez que um novo valor é colocado no final do array, área de atuação do algoritmo diminui em uma casa, portanto o segundo valor removido estará na penúltima posição do array. Isso é feito sucessivas vezes até que o array seja ordenado.

Sua implementação foi extremamente desgastante. A árvore do tipo heap possui uma lógica de construção curiosa, mas de difícil compreensão. A forma como os valores são removidos e ela se reconstrói recursivamente é genial, mas explicar isso com palavras de fácil compreensão é complicado.Ele possui: no pior caso $O(nlogn)$, no caso médio $O(nlogn)$ e melhor caso $O(nlogn)$. Confira sua tabela de dados 9

10. Gnome Sort

É um algoritmo de ordenação que se baseia numa ideia muito parecida com o bubble sort. Ele funciona da seguinte maneira: Comparo dois valores, um valor e seu antecessor, caso o valor seja menor que o seu antecessor, ocorre uma troca de lugar e o algoritmo retrocede uma casa para trás. Caso o valor seja maior que o seu antecessor o algoritmo avança uma casa para frente. Isso faz com que esse algoritmo percorra o array indo e voltando por várias e várias vezes. Confira a figura 11.

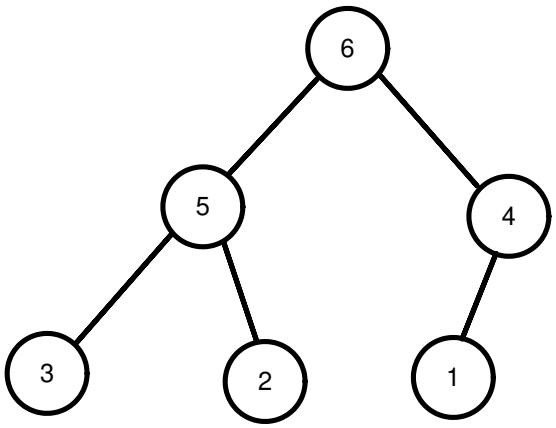

Figura 10. Heap Sort Árvore

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	32	33
100	00:00:00:00	2182	1881
1.000	00:00:00:00	34962	28665
10.000	00:00:00:00	483471	387381
50.000	00:00:00:00	2884509	2287305
100.000 (caso médio)	00:00:00:00	6169643	4875336
100.000 (pior caso)	00:00:00:00	5845553	4643541
100.000 (melhor caso)	00:00:00:00	6481489	5095836
500.000	00:00:00:06	35444098	27823020

Tabela 9. Heap Sort

Sua implementação é simples e não causou problemas. Sua lógica é parecida com o bubble sort, a diferença é que sempre uma troca ocorre ele volta uma casa para trás. Possui uma estrutura de funcionamento bastante compreensível, sem grandes surpresas. Ele possui no pior caso $O(n)$, no caso médio $O(n)$ e no melhor caso $O(n)$. Configura sua tabela de dados 10.

11. Radix Sort

É um algoritmo que se baseia na ocorrência dos dígitos. A versão do Radix Sort implementada foi a LSD(Dígito Menos Significante). Ele funciona pegando os dígitos, de trás pra frente, anota a sua ocorrência, ordena com base nessa ocorrência. E assim vai se repetindo de trás para frente. Como Ele se baseia num dígito específico, diferentemente do COunting SOrt, o seu arranjo de ocorrências possui no máximo 10 posições. Confira a figura 12.

Sua implementação foi caótica. Ele, em condições normais, foi feito para lidar com números positivos, então foi necessário encontrar uma solução para lidar com os números negativos durante a ordenação. Para fazer, um array com números positivos e outro com números negativos foram utilizados. Ele possui no pior caso $O(nk)$, no caso médio $O(nk)$ e no melhor caso $O(nk)$. Confira sua tabela de dados 11.

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	15	24
100	00:00:00:00	7323	9950
1.000	00:00:00:00	749940	1001902
10.000	00:00:00:02	75839778	101139690
50.000	00:00:16:04	1875046779	2500162340
100.000 (caso médio)	00:00:59:01	7512159393	10016412504
100.000 (pior caso)	00:01:54:05	14999351271	19999270930
100.000 (melhor caso)	00:00:00:00	0	199998
500.000	00:43:55:08	187519374846	250026833108

Tabela 10. Gnome Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	53	5
100	00:00:00:00	1009	100
1.000	00:00:00:00	10016	1000
10.000	00:00:00:00	100013	10000
50.000	00:00:00:00	500026	50000
100.000 (caso médio)	00:00:00:00	1000022	100000
100.000 (pior caso)	00:00:00:00	1016062	100000
100.000 (melhor caso)	00:00:00:00	1015989	100000
500.000	00:00:00:01	5000025	500000

Tabela 11. Radix Sort

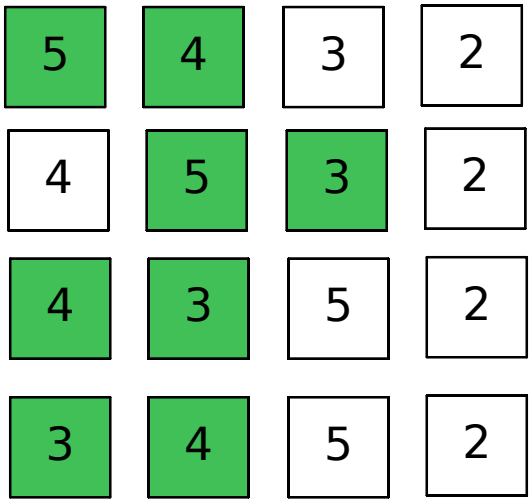

Figura 11. Gnome Sort

12. Counting Sort

É um algoritmo de ordenação que se utiliza de uma estratégia de ordenação completamente diferente dos anteriores nesse artigo. Na prática ele faz a ordenação sem comparar os valores uns com os outros. Para fazer isso existem dois arranjo, um arranjo principal com os valores a serem ordenados e um arranjo de contagem que conta a frequência dos valores armazenados no arranjo principal. Em suma, os valores no arranjo principal serão os índices no arranjo de contagem. Para lidar especificamente com valores negativos, o tamanho do arranjo de contagem é igual a $maiorValor - menorValor + 1$. A relação entre esses dois arranjos é igual a $valor - menorValor$, de modo que para encontrar a posição de um valor no arranjo de contagem eu devo subtrair esse valor ao menor valor no arranjo e assim eu terei a posição desse valor no arranjo de contagem. Logo em seguida, com a fórmula $indice + menor$ é aplicada ao arranjo de contagem e os valores reais são postos de maneira ordenada no arranjo. Confira a figura 13.

Sua implementação é muito difícil. Foi necessário um bom tempo para que a sua lógica fosse implementada e explicada. A construção do array de contagem seguida da sua desconstrução de maneira ordenada é genial e extremamente efetiva. Tudo é ordenada sem que nenhuma comparação seja feita. Ele possui no pior caso $O(n+k)$, no caso médio $O(n+k)$ e no melhor caso $O(n+k)$. Confira sua tabela de dados 12.

13. Bucket Sort

É um algoritmo de ordenação que separa os elementos em baldes individuais e depois os ordena separadamente dentro de cada balde. Primeiro você separa um intervalo de valores e segundo esse intervalo os números são peneirados em seus baldes individuais, depois, dentro dos baldes um algoritmo de ordenação qualquer é utilizado para ordená-los. Logo em seguida os baldes são despejados e o arranjo está completamente ordenado. Confira a figura 14.

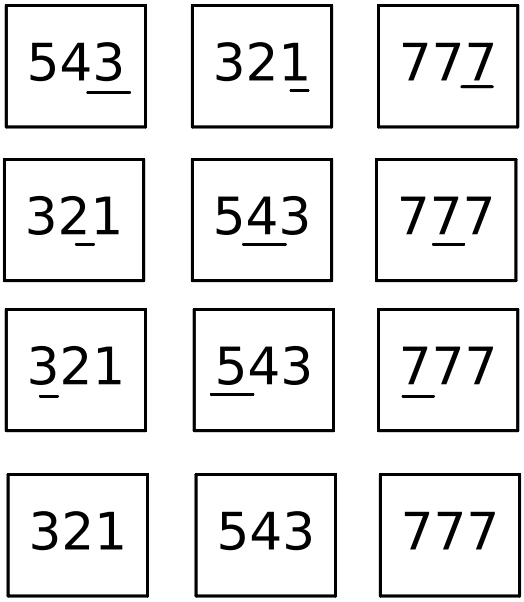

Figura 12. Radix Sort

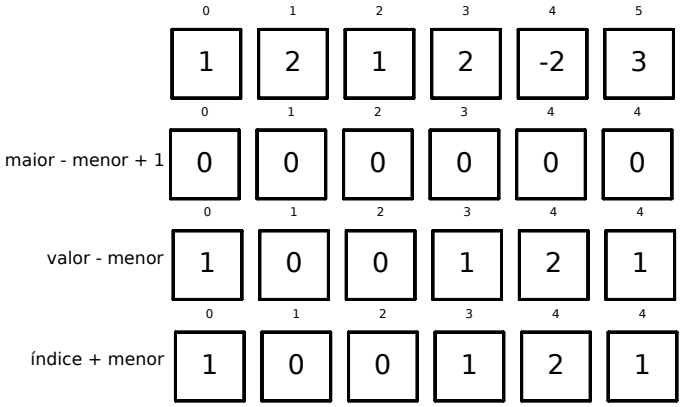

Figura 13. Counting Sort

14. Cocktail Sort

É um algoritmo de ordenação conhecido como Double Bubble Sort. Ele possui esse nome porque ele realiza um Bubble Sort indo da esquerda para a direita e logo em seguida outro Bubble Sort indo da direita para a esquerda. Em sua iteração, ele coloco o menor e o maior valor nas suas respectivas posições. Confira a figura 15.

Sua implementação ocorreu sem maiores problemas. É só uma questão de adapatar a lógica do Bubble Sort para ir e voltar. Ele possui no pior caso $O(n)$, no caso médio $O(n)$ eno melhor caso $O(n)$. Confira sua tabela de dados 13.

15. Tim Sort

É um algoritmo de ordenação extremamente rápido e novo. Ele se baseia na fusão de dois algoritmos diferentes: Insertion Sort e Merge Sort. Primeiro você separa o seu arranjo de

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	5	0
100	00:00:00:00	100	0
1.000	00:00:00:00	1000	0
10.000	00:00:00:00	10000	0
50.000	00:00:00:00	50000	0
100.000 (caso médio)	00:00:00:00	100000	0
100.000 (pior caso)	00:00:00:00	100000	0
100.000 (melhor caso)	00:00:00:00	100000	0
500.000	00:00:00:00	500000	0

Tabela 12. Counting Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	15	12
100	00:00:00:00	7323	4015
1.000	00:00:00:00	749940	378235
10.000	00:00:00:00	75839778	37805670
50.000	00:00:02:04	1875046779	940778985
100.000 (caso médio)	00:00:09:08	7512159393	3749424922
100.000 (pior caso)	00:00:06:00	14999351271	5000050000
100.000 (melhor caso)	00:00:00:00	0	100000
500.000	00:05:11:02	187519374846	93722118672

Tabela 13. Cocktail Sort

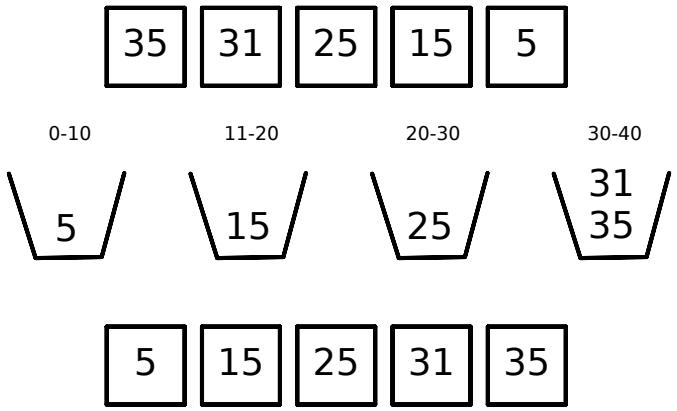

Figura 14. Bucket Sort

Qtd. de Números	Tempo de Exec.	Movimentos	Comparações
5	00:00:00:00	13	0
100	00:00:00:00	2334	246
1.000	00:00:00:00	34598	5753
10.000	00:00:00:00	427602	95576
50.000	00:00:00:00	2373499	593151
100.000 (caso médio)	00:00:00:00	4948083	1286841
100.000 (pior caso)	00:00:00:00	5179911	731601
100.000 (melhor caso)	00:00:00:00	2973452	684224
500.000	00:00:00:00	26791293	7461330

Tabela 14. Tim Sort

números em runs. Cada uma dessas runs são ordenadas utilizando o Insertion Sort e logo em seguida elas são unidas duas a duas através do Merge Sort. Confira a figura 16.

Sua implementação foi, de todos até aqui, a mais complicada. Sua lógica de funcionamento é bastante complexa e abstrata. Foi necessário a leitura de um artigo e o estudo de um vídeo passo-a-passo para a sua implementação e compreensão. Ele possui no pior caso $O(nlogn)$, no caso médio $O(nlogn)$ e no melhor caso $O(n)$. Confira sua tabela de dados 14.

Referências

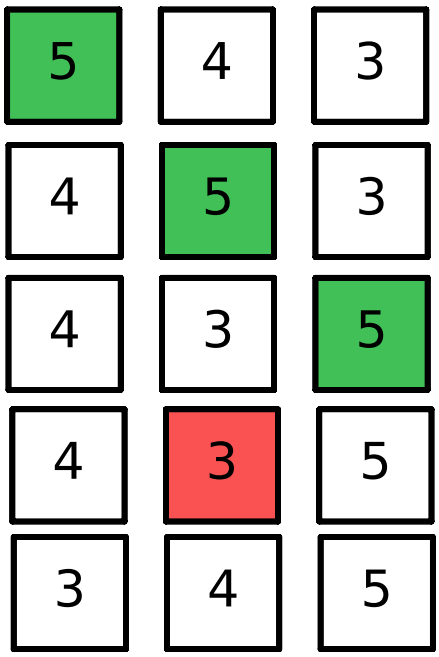

Figura 15. Cocktail Sort

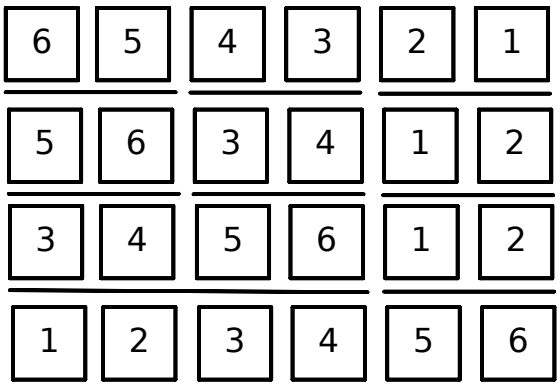

Figura 16. Tim Sort