Aplicação algorítmica de ordenação CountingSort e E-CountingSort

Kaique de Souza Leal Silva[1], Matheus Luz de Faria[1], Matheus Martins Batista[1], Thais Danieli Branco de Souza[1], Waldomiro Barbosa Romão[1]

[1]Instituto de Matemática e Computação – Universidade Federal de Itajubá (UNIFEI)
Caixa Postal 50 – 37500-903 – Itajubá – MG – Brasil

{d2020032426,leal.k,matmb,thaisdesouza,waldomiroromao}@unifei.edu.br

Abstract. *This article aims to analyze and implement the CountingSort and Efficient CountingSort ordering method in C programming language. Throughout the article, images were made available that illustrate the behavior of the algorithms, as well as theoretical foundation that details their implementation. Based on the understanding obtained from the analysis, one can define the advantages and disadvantages of using Countingsort algorithms.*

Resumo. *Este artigo tem como objetivo analisar e implementar o método de ordenação CountingSort e E-CountingSort na linguagem de programação C. Ao longo do artigo, foram disponibilizados a base dos algoritmos supracitados, seu comportamento, além de fundamentação teórica que detalha sua implementação. Com base no entendimento obtido a partir da análise, pode-se definir as vantagens e desvantagens do uso dos algoritmos de CountingSort.*

1. Introdução

Os algoritmos de ordenação são utilizados para se organizar um conjunto de dados, são conhecidos como Métodos de Ordenação ou Algoritmos de Ordenação. A título de exemplo, os métodos de ordenação são os de ordenação interna e ordenação externa. Dessa forma, entende-se como método de Ordenação Interna quando todos os elementos a serem ordenados cabem na memória principal e qualquer registro pode ser imediatamente acessado. Já os de Ordenação Externa é quando os elementos a serem ordenados não cabem na memória principal e os registros são acessados sequencialmente ou em grandes blocos. O método de Ordenação Interna em que se baseia esse artigo é o CountingSort, que estabelece sua ordenação por meio de contagens cuja complexidade é O(n).

O CountingSort determina que, para cada elemento de entrada x, o número de elementos deve ser menor do que x. Ele usa essas informações para colocar o elemento x diretamente em sua posição na matriz de saída. Por exemplo, se 17 elementos são menores que x, então x pertence à saída posição 18 [Cormen 2009]. O CountingSort é muito utilizado em sistemas que dependem do processamento de dados, visto que esse tipo de algoritmo possui rapidez e é de natureza estável, o que garante sua eficiência. Em busca de ampliar sua eficiência o E-CoutingSort possui melhorias em comparação ao algoritmo original, pelo fato de possuir uma complexidade de tempo reduzida em aproximadamente metade do original. Uma propriedade importante é que o CountingSort é estável, ou seja, números com o mesmo o valor aparecem na matriz de saída na mesma ordem que na matriz de entrada. A estabilidade do algoritmo CountingSort é importante por outro motivo:

é frequentemente usado como uma subrotina no algoritmo do RadixSort, pois para que ele possa funcionar corretamente o CountingSort deve ser estável [Cormen 2009].

O objetivo principal deste artigo é analisar e implementar o método de ordenação CountingSort e seu método eficiente E-CountingSort, a fim de analisar seus desempenhos de ordenação. Dessa forma, a constatar-se-á a base dos algoritmos evidenciando suas principais características de ordenação.

Em vista dos fatos supracitados, esse artigo busca implementar esse método de ordenação e estudar uma modificação proposta em [Bajpai and Kots 2014] que permita que esse algoritmo seja mais eficiente.

2. Referêncial Téorico

O algoritmo CountingSort utiliza 3 fors para registrar a frequência, o acumulativo e preencher um vetor de saída ordenado. Nesse Contexto, considere a Figura 1 fornecida do livro [Cormen 2009] para a seguinte explicação.

Após o loop for das linhas 2–3 inicializar o array C para todos os zeros, o loop for das linhas 4–5 inspeciona cada elemento de entrada. Se o valor de um elemento de entrada é i, incrementamos C[i]. Assim, após a linha 5, C[i] vale o número de elementos de entrada igual a i para cada inteiro i = 0, 1, ..., k. As Linhas 7-8 determinam para cada i = 0, 1, ..., k quantos elementos de entrada são menores ou iguais para i mantendo uma soma corrente do array C. Finalmente, o laço for das linhas 10–12 coloca cada elemento A[j] em seu correto posição ordenada na matriz de saída B.

Figura 1. Implementação do Cormen do algoritmo CountingSort.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Para fins de implementação, algumas mudanças foram feitos no algoritmo fornecido pelo livro, conforme a Figura 2. Nessa modificação do CountingSort, levou-se em consideração que os elementos nos vetores ou arrays começam em zero, para concretizar a correta implementação do algoritmo. Dessa forma, os loops (for) não mais saiam de um índice 1, mas sim de um índice 0, indicando a primeira posição do vetor.

O artigo proposto [Olhar referência] propõe uma versão eficiente do CountingSort sem o loop para fazer o valor acumulativo, o E-CountingSort aproveito os índices dos elementos presente no vetor de contagem e o frequência para inseri-los N vezes no vetor de saída ordenado, sendo N a frequência do índice de um elemento do vetor de contagem.

Figura 2. Implementação do Cormen de CountingSort modificada.

Counting Sort Algorithm

```
COUNTING_SORT (A, B, k, n)

1.      for i ← 0 to k do
2.          C[i] ← 0
3.      for j ← 0 to n do
4.          C[A[j]] ← C[A[j]] + 1
5.      for i ← 1 to k do
6.          C[i] ← C[i] + C[i-1]
7.      for j ← (n - 1) down to 1 do
8.          B[C[A[i]] - 1] ← A[j]
9.          C[A[i]] ← C[A[j]] - 1
```

Para inserir esse elementos N vezes, um loop while verifica se há frequência no elemento de índice i do vetor de contagem, se houver insere no vetor de saída ordenada e decresce a frequência até que ela seja 0. A Figura 3 ilustra essa ideia.

Figura 3. CountingSort Eficiente

E-COUNTING SORT ALGORITHM

```
COUNTING_MODIFY_SORT (A, B, k)

1.      j=0
2.      for i← 0 to k
3.          C[i] ←0;
4.      for i← 0 to n
5.          C[a[i]]=C[a[i]]+1;
6.      for i← 0 to k
7.          while(c[i]>0)
8.              a[j]=i;
9.              j=j+1;
10.             c[i]=c[i]-1
```

3. Metodologia

A metodologia utilizada partiu do estudo e revisão do artigo e livro disponibilizados. Neles foram encontradas explicações sobre o código do algoritmo CountingSort, que foram

fundamentais para a sua aplicação e montagem. Para validar a sua correta lógica, fez-se um teste de mesa. O código feito segue na Figura 4:

Figura 4. Implementação do CountingSort.

```c
void countingSort(int *arrA, int *arrB, int max, int tam, data *comp){
    int i;
    int* count = calloc(max + 1, sizeof(int));

    for (i = 0; i < tam; i++)
    {
        count[arrA[i]]++;
        comp->copy++;
    }

    for (i = 1; i <= max; i++)
    {
        count[i] += count[i - 1];
        comp->copy++;
    }

    for (i = tam - 1; i >= 0; i--)
    {
        arrB[count[arrA[i]] - 1] = arrA[i];
        count[arrA[i]]--;
        comp->copy++;
    }
}
```

Primeiramente, criou-se a função countingSort os seguintes parâmetros foram fornecido: vetor que será ordenado, um vetor de saída ordenado, o maior elemento, tamanho do vetor e uma estrutura para colher as métricas. O algoritmo fornecido pelo livro supracitado zera as posições do vetor de contagem após aloca-lo, para fins práticos foi utilizado a função calloc() da biblioteca stdlib. h na linguagem C (aloca memória e inicializa os campos com 0). Em seguida, fez-se a declaração do primeiro for que tem como objetivo registrar a frequência. O segundo for, tem como objetivo guardar o acumulativo dos valores menores que um elemento i do vetor original (A). Por fim, a função é fechada por um último for que tem como objetivo percorrer o vetor original com início no último elemento, substituindo os índices nos elementos do vetor count e decrescendo a cada atribuição. Vale ressaltar que para ambos os loops, variáveis de registros foram incrementadas a fin de calcular a média das execuções.

De modo similar ao CountingSort, o E-CountingSort foi definido pela função eCountingSort e suas variáveis. O primeiro for, utiliza o vetor count para guardar a frequência. Ademais, um for que diferencia a função do E-CountingSort, pelo fato de ter como objetivo aproveitar a frequência dos elementos e o índice do vetor count para preencher o vetor B ordenado. O código feito segue na Figura 5:

Figura 5. Implementação do CountingSort Eficiente.

```c
void eCountingSort(int *arrA, int *arrB, int max, int tam, data *d){
    int i, j = 0;
    int* count = calloc(max + 1, sizeof(int));

    for (i = 0; i < tam; i++){
        count[arrA[i]]++;
        d->copy++;
    }

    for (i = 0; i <= max; i++){
        while(count[i] > 0){
            arrB[j] = i;
            j++;
            count[i]--;
            d->copy++;
            d->comp++;
        }
    }
}
```

4. Resultados

Essa sessão irá discutir os resultados obtidos no trabalho.

4.1. Máquina utilizada para o processamento dos dados

A máquina utilizada para os testes tem as seguintes configurações:

- Processador: Intel Core 5-9300HF @ 2.40GHz;
- Memória: 8,00 GB;
- Sistema Operacional: Windows 11 Home Single
- Compilador: gcc version 6.3.0 (MinGW.org GCC-6.3.0-1);
- Comando: gcc -Wall main.c countingSort.h countingSort.c -o ./count.exe;
- Método de execução: Windows PowerShell.

4.2. Registro de saída

Para meios de comparação e exposição dos resultados, o programa gera um arquivo de registro que guarda os casos para os valores de N fornecidos de cada tipo de vetor. Dessa forma, fez-se um arquivo de saída que exibe os resultados obtidos pelos algoritmos para ser analisada, que utilizaram dos mesmos dados. Para o caso de comparação, fez-se o uso de um milhão de números. Seguem os dados de saída utilizados na Figura 6.

Figura 6. Dados obtidos para análise

	COUNTING SORT	ECOUNTING SORT	COUNTING SORT	ECOUNTING SORT	COUNTING SORT	ECOUNTING SORT
Copy	Aleatórios	Aleatórios	Ordenados	Ordenados	Decrescente	Decrescente
N = 10000	49995	20000	49995	20000	49995	20000
N = 100000	229999	200000	229999	200000	229999	200000
N = 500000	1029999	1000000	1029999	1000000	1029999	1000000
N = 1000000	2029999	2000000	2029999	2000000	2029999	2000000
Comp						
N = 10000	0	10000	0	10000	0	10000
N = 100000	0	100000	0	100000	0	100000
N = 500000	0	500000	0	500000	0	500000
N = 1000000	0	1000000	0	1000000	0	1000000
Time						
N = 10000	0,0003	0,0005	0,0004	0,0001	0,0002	0,0002
N = 100000	0,0012	0,001	0,0009	0,0012	0,0011	0,001
N = 500000	0,0065	0,0041	0,0047	0,0041	0,0045	0,0039
N = 1000000	0,0121	0,0076	0,0094	0,0079	0,0088	0,0078

Abaixo, segue o link para o vídeo onde a execução do algoritmo e do arquivo de saída são realizadas. Ademais, o link para acesso do código no GitHub.

- $https://www.youtube.com/watch?v = Xo1Jk8XnkBo$;
- $https://github.com/Maysu115/trab_aed_I_counting_counting$.

4.3. Gráficos

O gráfico a seguir representa os dados do teste realizado com todos os valores de N propostos (10000,100000,500000,1000000), sendo disposto na plotagem o número de cópias, comparações e o tempo de execução:

A Figuras 7, 8, 9 representam as execuções medindo a quantidade de cópias de registro geradas.

Pode-se afirmar que o CountingSort faz muito mais registros que o eficiente.

Figura 7. Comparação entre CountingSort e E-CountingSort para número de cópias de vetor ordenado

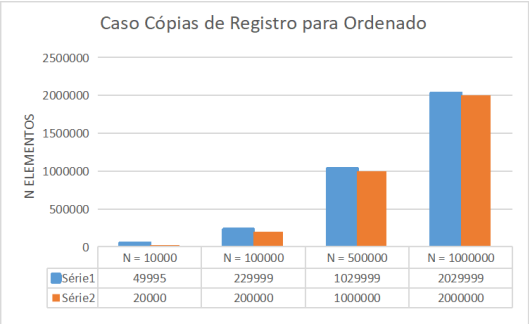


Figura 8. Comparação entre CountingSort e E-CountingSort para número de cópias de vetor aleatório

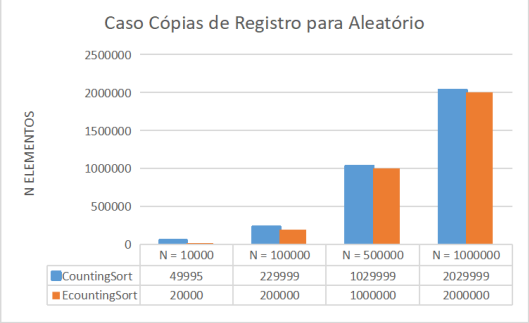
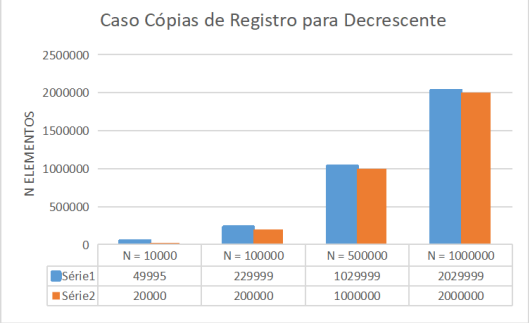


Figura 9. Comparação entre CountingSort e E-CountingSort para número de cópias de vetor decrescente



A Figuras 10, 11, 12 representam as execuções medindo a quantidade de comparações geradas.

Pode-se afirmar que o CountingSort é um algoritmo que não faz comparações, utiliza apenas das posições dos valores para ordenar inteiros. Em contraste, o E-CountingSort precisa faz as comparações (verificar se a frequência é menor que 0, ou seja, não há mais elementos de valor igual ao índice buscado) para preencher o vetor de saída com as frequências dos números que aparecem no vetor de entrada.

Figura 10. Comparação entre CountingSort e E-CountingSort para número de comparações de vetor ordenado

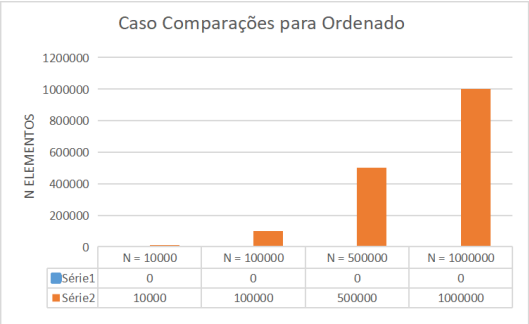


Figura 11. Comparação entre CountingSort e E-CountingSort para número de comparações de vetor aleatório

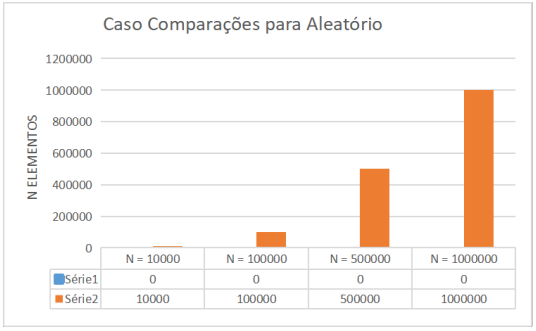
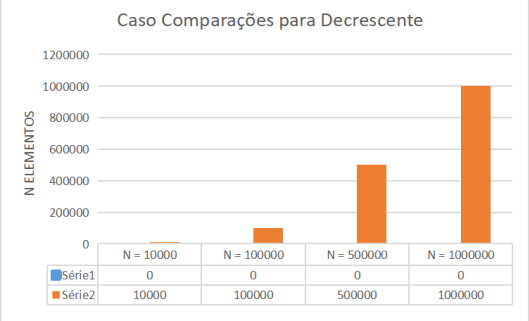


Figura 12. Comparação entre CountingSort e E-CountingSort para número de comparações de vetor decrescente



A Figuras 13, 14, 15 representam as execuções medindo a quantidade de comparações geradas.

Pode-se afirmar que em tempo e simplicidade de execução o E-CountingSort tem um melhor desempenho. Ou seja, ele executa em menos tempo na maioria dos casos para N valores fornecidos.

Figura 13. Comparação entre CountingSort e E-CountingSort para tempo de execução de vetor ordenado

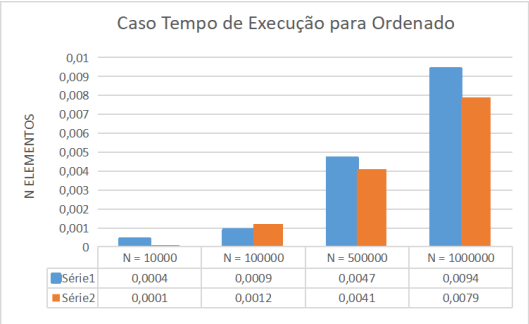


Figura 14. Comparação entre CountingSort e E-CountingSort para tempo de execução de vetor aleatório

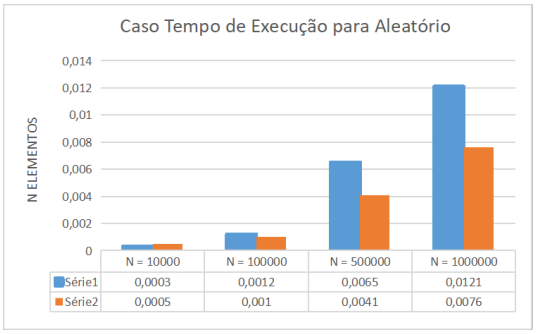
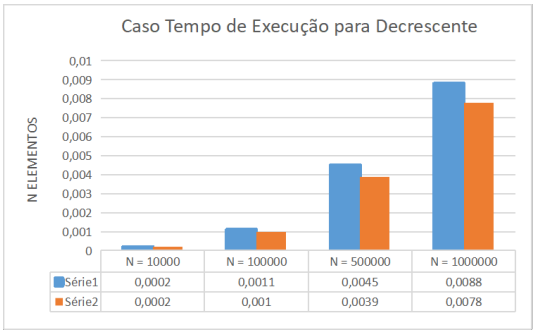


Figura 15. Comparação entre CountingSort e E-CountingSort para tempo de execução de vetor decrescente



4.4. Discussão e análise dos resultados

Notou-se, por meio do arquivo de saída, que o número médio de cópias do algoritmo do E-CountingSort foi notavelmente menor, assim como o tempo de execução (mesmo que próximo), em relação ao CountingSort, para pequenas quantidades de dados. Entretanto, para grandes quantidades, nota-se uma diferença grotesca no tempo de execução, o qual foi denotado o menor pelo algoritmo de E-CountingSort, comprovando sua melhor eficiência em relação ao CountingSort.

5. Considerações finais

Entende-se, portanto, que o algoritmo do CountigSort possui vantagens, como a de ser estável, por não alterar a ordem dos dados, além de possuir um processamento simples.

Em questão de desempenho o E-CountingSort proposto é realmente melhor que CountingSort. Todabia os ambos os casos estão limitados para números inteiros. De fato nota-se desvantagens acerca de sua limitação com o manuseio de grandes conjuntos de dados, considerando k o valor do maior elemento, se este for muito grande, será alocada uma quantidade considerável de memória para o vetor de contagem, contudo podem haver diversas posições desse vetor que ficaram vazias do ínicio ao fim da execução dos algoritmos supracitados.

O seu aperfeiçoamento, denominado E-CountingSort, faz com que o manuseio de grandes conjuntos de dados melhore seu tempo de execução, como observado nos resultados dispostos, comprovando o porquê de sua eficiência. Considerando uma complexidade O(n + k), para o CountingSort é necessário analisar os 3 loops (de complexidade n,k e n, respectiamente), já para o E-CountingSort são dois loops a serem analisados (n e k). Ou seja, para os casos onde n é maior que k, a eficiência do algoritmo é garantida.

Referências

Bajpai, K. and Kots, A. (2014). Implementing and analyzing an efficient version of counting sort (e-counting sort). In *International Journal of Computer Applications*, volume 3, pages 1–2. IJCA.

Cormen, T. H., e. a. (2009). Countingsort. In *Introduction to algorithms*, pages 194–197. The MIT press, 3 edition.