

LuFood: Implementação do Counting Sort para Ordenação de Pedidos no Sistema LuFood

**Levi Falcão de Queiroz, Gabriel Gomes Cruz Uzeda, Bruno Sampaio Silva,
Claudio dos Santos Junior, Herick Marcio Matos Brito, Breno Matos Bastos**

Centro Universitário de Excelência (UNEX) - Sistemas de Informação - Feira de Santana - Bahia - Brasil

{leviq.f@hotmail.com, gomes.uzeda@gmail.com, herickmarcio356@gmail.com,
brunosilva.mee@gmail.com, brenomatos486@gmail.com,
claudiosjrel@gmail.com}

Resumo. Este artigo detalha a refatoração do sistema de delivery, com foco na implementação do algoritmo Counting Sort. O objetivo primário é desenvolver um método eficiente para a ordenação de pedidos finalizados a partir de seu código numérico sequencial. Descrevemos a adaptação do Counting Sort, tradicionalmente usado para inteiros, para operar sobre uma estrutura de dados complexa. Os resultados da implementação validam a corretude do algoritmo e estabilidade.

Abstract. This paper details the refactoring of the delivery system , focusing on the implementation of the Counting Sort algorithm. The primary objective is to develop an efficient method for sorting finalized orders based on their sequential numeric code. We describe the adaptation of the Counting Sort, traditionally used for integers, to operate on a complex data structure. The implementation results validate the algorithm's correctness and stability

1. Introdução

Sistemas de informação modernos são definidos por sua capacidade de gerenciar vastos conjuntos de dados. A ordenação é uma das operações mais fundamentais nesse gerenciamento. O "problema de ordenação" é vasto, com inúmeros algoritmos disponíveis, cada um com suas vantagens e desvantagens. O desafio não é simplesmente ordenar, mas escolher a ferramenta certa para o trabalho. No escopo do projeto "Tia Lu", a necessidade de ordenar pedidos por um código numérico sequencial apresenta um cenário ideal para algoritmos não-comparativos.

2. Fundamentação Teórica

A ordenação é um processo fundamental na ciência da computação. Enquanto algoritmos de ordenação baseados em comparação, como o Quick Sort ou Merge Sort, possuem uma complexidade de tempo média de $O(n \log n)$, existem cenários específicos onde algoritmos não-comparativos podem atingir performance superior

2.1 Características do Counting Sort

O Counting Sort é um algoritmo **não-comparativo**; ele não compara elementos entre si ($A > B$) [Programiz 2025]. Sua eficiência deriva de contar a frequência de cada chave. Além disso, quando implementado corretamente, é um algoritmo **estável**: elementos com a mesma chave mantêm sua ordem relativa original após a ordenação.

A figura 1 representa o pseudo código contido na função counting_sort() é:

```
valor_maximo = 0
PARA CADA item EM lista:
    SE item[chave_id] > valor_maximo:
        valor_maximo = item[chave_id]

tamanho_lista = comprimento(lista)
lista_contagem = criar_lista_de_tamanho(valor_maximo + 1) preenchida com 0
lista_ordenada = criar_lista_de_tamanho(tamanho_lista)

PARA i DE 0 ATÉ tamanho_lista - 1:
    elemento = lista[i]
    valor_chave = elemento[chave_id]
    lista_contagem[valor_chave] = lista_contagem[valor_chave] + 1

PARA i DE 1 ATÉ comprimento(lista_contagem) - 1:
    lista_contagem[i] = lista_contagem[i] + lista_contagem[i - 1]

i = tamanho_lista - 1
ENQUANTO i >= 0:
    elemento = lista[i]
    valor_chave = elemento[chave_id]

    posicao_saida = lista_contagem[valor_chave] - 1
    lista_ordenada[posicao_saida] = elemento
    lista_contagem[valor_chave] = lista_contagem[valor_chave] - 1
    i = i - 1

RETORNE lista_ordenada
```

Figura 1 Pseudocódigo.

3. Metodologia e Implementação

A implementação foi realizada na linguagem Python, utilizando um módulo dedicado (ordenacao.py) para encapsular a lógica, conforme as diretrizes de refatoração. O principal desafio metodológico foi adaptar o Counting Sort, que por definição opera sobre um array de inteiros, para ordenar uma lista de dicionários Python (mapas).

O funcionamento do nosso `counting_sort(lista_de_dicionarios, id)` segue a estratégia clássica do algoritmo, dividida em quatro fases de execução:

1. **Mapeamento:** O algoritmo varre a lista_de_dicionarios para encontrar o maior valor da chave (id) fornecida. Esse valor max define o tamanho do vetor auxiliar.
2. **Contagem (Frequência):** Um vetor auxiliar (count) de tamanho max + 1 é inicializado com zeros. O algoritmo itera pela lista de entrada e usa o valor_chave de cada dicionário para incrementar a posição correspondente no vetor count.
3. **Soma:** O vetor count é transformado para que cada posição i armazena a soma de todas as contagens anteriores até i (`count[i] += count[i - 1]`). Agora, `count[i]` representa a **posição final** do último elemento com chave i.
4. **Construção da Saída (Estabilidade):** O algoritmo cria a lista_ordenada e itera pela lista de entrada **de trás para frente**. Para cada elemento, ele consulta sua posição final em count, insere o elemento na lista_ordenada e decremente o valor em count. Iterar de trás para frente é o que garante a **estabilidade**.

Na figura abaixo podemos ver uma representação do nosso algoritmo. contendo a lista a ser ordenada, a lista “auxiliar” e o output

Input Data														
0	4	2	2	0	0	1	1	0	1	0	2	4	2	
Count Array														
0	1	2	3	4										
5	3	4	0	2										
Sorted Data														
0	0	0	0	0	1	1	1	2	2	2	2	4	4	

Figura 2 Counting Sorting Exemplo.

A solução de parametrizar a chave de ordenação (id) tornou a função modular e reutilizável, pronta para ser aplicada em outras entidades do sistema (como "itens" por "estoque", por exemplo).

4. Resultados

Para validar a implementação, executamos o script (`ordenacao.py`) com uma massa de dados estática. O conjunto de teste incluía 5 pedidos com códigos desordenados e, crucialmente, códigos duplicados ("codigo: 5") para testar o algoritmo.

4.1 Funcionamento

A **Figura 3** apresenta o *output* do console antes da aplicação do algoritmo, mostrando a lista de pedidos na ordem original de inserção.

```
--- Lista Desordenada ---  
{'codigo': 5, 'status': 'ENTREGUE', 'valor_final': 100.0}  
{'codigo': 2, 'status': 'ACEITO', 'valor_final': 50.0}  
{'codigo': 8, 'status': 'FAZENDO', 'valor_final': 75.0}  
{'codigo': 1, 'status': 'AGUARDANDO APROVACAO', 'valor_final': 120.0}  
{'codigo': 5, 'status': 'REJEITADO', 'valor_final': 30.0}
```

Figura 3 - Lista desordenada

A **Figura 4** demonstra o *output* do console após a execução da função `counting_sort(pedidos_teste, "codigo")`.

```
--- Lista Ordenada ---  
{'codigo': 1, 'status': 'AGUARDANDO APROVACAO', 'valor_final': 120.0}  
{'codigo': 2, 'status': 'ACEITO', 'valor_final': 50.0}  
{'codigo': 5, 'status': 'ENTREGUE', 'valor_final': 100.0}  
{'codigo': 5, 'status': 'REJEITADO', 'valor_final': 30.0}  
{'codigo': 8, 'status': 'FAZENDO', 'valor_final': 75.0}
```

Figura 4 - Lista ordenada pela função `counting_sort()`

4.2 Análise

A análise da Figura 4 prova que o algoritmo ordenou corretamente os pedidos pela chave "codigo" (1, 2, 5, 5, 8). O ponto mais crítico da validação é a estabilidade, pois o mesmo está em uma função, e caso mal projetada pode resultar na quebra do código.

5. Considerações finais

A implementação do Counting Sort foi concluída com sucesso, provando ser uma ferramenta de extrema eficiência para a necessidade específica do projeto de ordenar pedidos por um código numérico único de cada pedido ou outro dado

O desafio central foi, sem dúvida, a adaptação do algoritmo para uma estrutura de dados moderna (dicionários). A solução de parametrizar a chave de ordenação (id) tornou a função modular e reutilizável, pronta para ser aplicada em outras entidades do sistema (como "itens" por "estoque", por exemplo).

Melhorar o algoritmo *Counting Sort* para suportar chaves negativas consiste em adaptar sua estrutura de contagem, que originalmente assume apenas valores inteiros não negativos.

6. Referências

PROGRAMIZ. (2025). *Counting Sort (With Code in Python/C++/Java/C)*. Disponível em:
<https://www.programiz.com/dsa/counting-sort>.

THECRAZYPROGRAMMER. (2015). Counting Sort Program in C. Disponível em:
<<https://www.thecrazyprogrammer.com/2015/04/counting-sort-program-in-c.html>>. Acesso em:
16 nov. 2025.