COUNTING SORT E PIGEONHOLE SORT

Algoritmos de Ordenação de Dados

1st Kauã Silva
Engenharia da Computação
CEFET MG Campus V
Oliveira MG, Brasil
kaualucas396@gmail.com

2nd Humberto Cunha
Engenharia da Computação
CEFET MG Campus V
Itapecerica MG, Brasil
humberto17henrique@gmail.com

3rdAnderson Santos
Engenahria da Computação
CEFET MG Campus V
Taiobeiras MG, Brasil
andersonifnmg.info@gmail.com

- Reusmo: Os algoritmos de ordenação são fundamentais em ciência da computação, desempenhando um papel crucial na organização e análise de dados. Entre as diversas técnicas disponíveis, o Counting Sort e o Pigeonhole Sort se destacam por sua eficiência em situações específicas. Ambos utilizam a contagem de elementos e a manipulação direta desses valores para ordenar listas, oferecendo alternativas viáveis aos métodos de comparação tradicionais.

- Palavras-chaves: Counting Sort, Pigeonhole Sort, Algoritmo, Ordenação

I. Introdução

No cenário das ciências da computação, o uso correto de algoritmos de ordenação destaca-se como um pilar fundametal para a aprendizagem e manipulação de estrtura de dados. Nesse contexto, surge o conceito de Análise Assintótica, o qual consiste na ideia de analisar como tais algoritmos lidam com entradas de diferentes tipos e tamanhos, a fim de se compreender o funcionamento dos mesmos e aprimorar o custo computacional de um programa.

Sob esse viés, este artigo explora dois dos diversos algoritmos de ordenação de dados existentes: o Counting Sort e o Pigeonhole Sort. Serão analisadas e trabalhadas as caracteríticas e aplicações destes algoritmos com diferentes tipos e tamanhos de entrada, detalhando suas metodologias, resultados, vantagens, desvantagens e casos ideiais de aplicação, possibilitando o uso adequeado e eficiente desses algoritmos pelos alunos, aprimorando ainda mais suas habilidades de programação. Primeiramente abordaremos o algoritmo Counting Sort e, em seguida, o Pigeonhole Sort.

II. Metodologia

Counting Sort:

A. Apresentação

O Counting Sort é um algoritmo de ordenação eficiente, não comparativo e out-of-place, desenvolvido por Harold H. Seward, um cientista da computação, engenheiro e inventor, em 1954 no Instituto de Tecnologia de Massachusetts. Harold propôs este algoritmo em um artigo intitulado ”Information Sorting in the Application of Electronic Digital Computers to Business Operations”, no qual abordou várias técnicas de ordenação.

A ideia básica por trás do Counting Sort é contar o número de ocorrências de cada valor único na entrada e usar essas contagens para determinar diretamente a posição de cada elemento na sequência ordenada. É particularmente eficiente quando o intervalo de valores dos elementos é conhecido antecipadamente e é relativamente pequeno comparado ao número total de elementos a serem ordenados.

Sua implementação utiliza uma lista de entrada (A), de tamanho n, com elementos que estão no intervalo de 0 a k, onde k é um número inteiro e positivo, representando o maior número da lista.

O algoritmo ainda possui a Lista de Contagem (C), de tamanho k+1, que fornece temporariamente o armazenamento das contagens cumulativas da Lista A. E a Lista de Saída (B), de tamanho N, é a saída ordenada.

B. Funcionamento

Para explicar o funcionamento do algoritmo Counting Sort, podemos analisar a explicação feita pelo autor Thomas H. Cormen no livro “Algoritmos: Teoria e Prática” (3ª edição americana). A ideia foi explicada de maneira implícita, mostrando que o algoritmo consiste em 3 etapas: Contagem, marcação de posição e ordenação.

Cormen utiliza como exemplo uma ordenação de uma lista de ordem crescente, onde a lista de entrada possui tamanho 8 e tem elementos que variam de 0 a 5, de forma desordenada.

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

Fig. 1. Lista de Entrada.

1) Contagem: Nesta primeira etapa, realiza-se a contagem dos elementos da lista de entrada A e os resultados são armazenados na lista C, onde cada posição corresponde ao número encontrado em A.

Na lista A, o número 0 aparece duas vezes. Portanto, na lista C, de tamanho k+1=6 (onde k=5), a primeira posição corresponde ao número 0 e deve ser incrementada em dois,

refletindo a contagem cumulativa dos números da lista de entrada A.

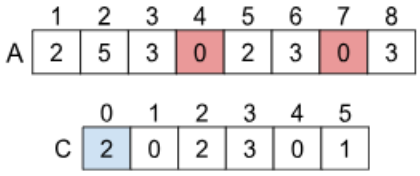

Fig. 2. Lista de Entrada.

2) Marcação de Posição: Na marcação de posição, cada posição na lista C é atualizada somando-se ao valor da posição anterior. Isso garante que cada posição em C represente a contagem acumulada dos elementos da lista A, refletindo a posição correta na lista ordenada.

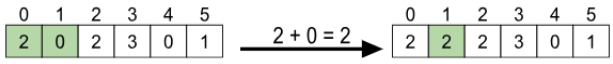

Fig. 3. A posição 1 de valor 0 somará com a posição anterior de valor 2.

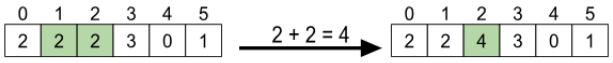

Fig. 4. A posição 2 de valor 2 somará com a posição anterior de valor 2.

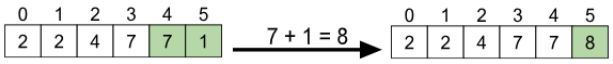

Fig. 5. Mostra a última alteração de somatória da posição 5 de valor 1 com a posição anterior de valor 7.

3) Ordenação: Nessa última etapa, ocorre a ordenação dos elementos da lista A utilizando os dados da lista C. A ordenação surge quando é feito o percurso da última posição para o primeiro de A, e cada valor do elemento em percurso será utilizado para buscar a posição determinante em C que será determinada a posição que o elemento de A ficará em B.

É importante notar que a ordem que esta ordenação é feita garante a ordenação correta dos elementos, mesmo que estes possuam um valor igual, corroborando a estabilidade do algoritmo.

Na figura 6, é demonstrado o processo de ordenação, onde o último elemento de A, que é o número 3, terá a posição determinada pelo index 3 da lista C (C[3] = 7), lembrando que a posição final será o elemento da casa do C[3] menos 1, devido ao ajuste necessário por causa do índice começando em 0. Ou seja, a posição final do último elemento de A será na posição B[7-1 = 6].

Na lista C, ocorrerá um reajuste de menos 1 para o valor acessado pelo elemento A. Ou seja, a posição C[3] de valor 7 passará a ser 6. Esse reajuste é essencial para que mantenha a estabilidade na ordenação dos elementos com chaves iguais, garantindo assim a precisão e eficiência do algoritmo.

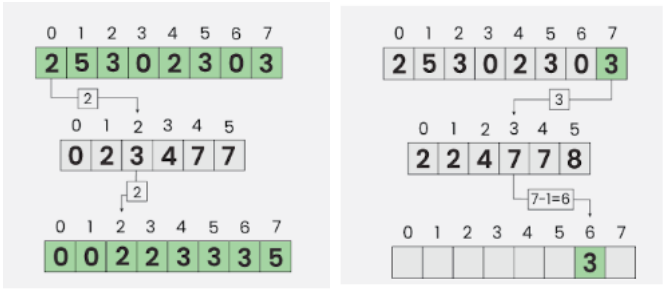

Fig. 6. As ilustrações A e B mostram o primeiro e o último processo de ordenação.

```
[6]  a = [2, 5, 3, 0, 2, 3, 0, 3]
     c = [0] * (max(a) + 1)

     # Passo 1: Contagem das ocorrências dos elementos
     for i in a:
         c[i] += 1

     # Passo 2: Determinação das posições finais
     for i in range(1,len(c)):
         c[i] += c[i - 1]

     # Passo 3: Construção do array ordenado
     b = [0] * len(a)
     for i in reversed(a):
         b[c[i] - 1] = i
         c[i] -= 1

     print(b)

     [0, 0, 2, 2, 3, 3, 3, 5]
```

Fig. 7. Mostra o algoritmo do exemplo descrito por Cormen na linguagem Python.

Pigeonhole Sort:

C. Apresentação

O Pigeonhole Sort é uma variação do Counting Sort, com a diferença de este algoritmo de ordenação é baseado no conceito dos ”buracos de pombo”, uma técnica eficiente para ordenar listas onde o número de elementos (n) e o intervalo de valores possíveis (N) são aproximadamente do mesmo tamanho. Assim como o Counting Sort, ele também é um algoritmo de ordenação eficiente, não comparativo, out-of-place e estável.

A ideia principal do Pigeonhole Sort é distribuir os elementos de entrada em ”gavetas” correspondentes ao valor desses elementos e, em seguida, coletar os elementos das gavetas em ordem sequencial.

As origens exatas do tipo Pigeonhole não são claras, pois é um algoritmo muito antigo que é conhecido há séculos. No entanto, foi formalmente descrito e analisado na literatura de ciência da computação por A. J. Lotka em 1926, e mais tarde redescoberto independentemente por John Knuth na década de 1960.

D. Funcionamento

O Pigeonhole Sort é um algoritmo de ordenação que utiliza uma abordagem simples e prática, ideal para conjuntos de dados onde os elementos estão dentro de um intervalo restrito. Baseia-se no princípio das gavetas (pigeonholes), que postula que se mais de n itens são colocados em n recipientes, pelo menos um recipiente conterá mais de um item.

Primeiro, identifica-se o valor mínimo e máximo da lista. Em seguida, cria-se um vetor de ”buracos” com tamanho igual à diferença entre o valor máximo e mínimo, mais um. Cada elemento da lista é colocado no ”buraco” correspondente ao seu valor. Por fim, os elementos são recolhidos dos ”buracos” em ordem, resultando em uma lista ordenada.

Podemos deomonstrar o funcionamento do Pigeonhole Sort em 4 etapas:

1) Determinar o tamanho do vetor: Nesta etapa, o algoritmo percorre a lista de valores de entrada e encontra o maior (max) e menor (min) valor, utilizando-os como base para criar um array com n posições (intervalo entre os valores). A lógica para a definição do tamanho do vetor é Max - Min + 1.

Exemplo: Para a lista A com valores [6, 3, 2, 7, 4, 8, 6]

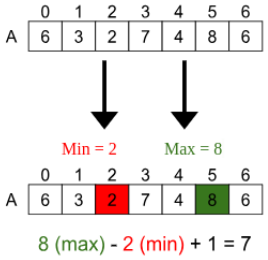

Fig. 8. Mostra o processo para determinar o tamanho do vetor.

Note que o vetor possui 7 posições, uma vez que o número de menor valor (2) foi subtraído do número de maior valor (8) e foi adicionada 1 nova posição, já que a primeira posição do vetor é sempre 0.

2) Criar os pigeonholes: Crie um array de pigeonholes (buracos de pombo). Cada índice deste array representará um valor possível na lista original. O tamanho do array de pigeonholes será igual ao intervalo (range) determinado no passo anterior.

Cada pigeonhole armazena a quantidade de vezes que determinado número aparece no vetor. Com isso, ele consegue saber quantas vezes certa posição foi preenchida por um valor e, então, saber quantos números repetidos existem no vetor, ordenando-os um ao lado do outro.

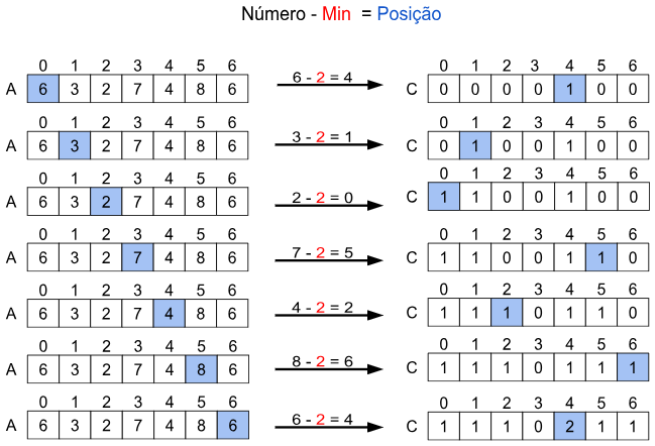

Fig. 9. Monta um vetor de pigeonholes armazenando em cada índice o número de vezes que cada algarismo aparece no vetor.

3) Coletar elementos e Converter os índices: Percorra os pigeonholes em ordem crescente de índice e colete os elementos, reconstruindo a lista ordenada. Esta reconstrução se dá por meio de um for que acessa cada posição do vetor e soma essa posição à entrada de menor valor, alocando cada valor em sua respectiva posição dentro do array

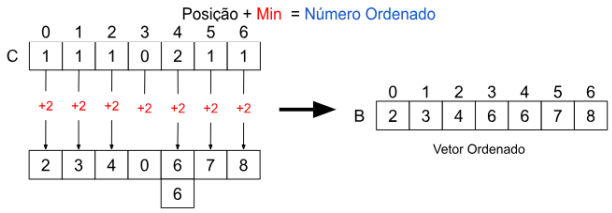

Fig. 10. Mostra o processo final para a criação do vetor ordenado

```python
a = [6, 3, 2, 7, 4, 8, 6]

# Encontrar o mínimo e o máximo no vetor
minimum = min(a)
maximum = max(a)

# Determinar o tamanho do vetor de Pigeonholes
size = maximum - minimum + 1

# Criar o vetor de Pigeonholes inicialmente com zeros
c = [0] * size

# Distribuir os elementos do vetor original nos Pigeonholes
for number in a:
    c[number - minimum] += 1

# Reconstruir o vetor ordenado a partir dos Pigeonholes
index = 0
for j in range(size):
    while c[j] > 0:
        a[index] = j + minimum
        index += 1
        c[j] -= 1

print(a)
```

```
[2, 3, 4, 6, 6, 7, 8]
```

Fig. 11. Algoritmo de Pigeonhole Sort

III. Vantagens e Desvantagens

Ambos os algortimos são parecidos, o que nos leva a pensar que suas vantagens e desvantagens também são semelhantes. Vale destacar que, pelo fato de serem algorimos não comparativos, não há necessidade de se comparar os elementos de um vetor, o que aprimora o tempo de execução, mas abre mão da memória, tendo em vista que ambos criam outro vetor para realizarem a ordenação, o que ocupa mais espaço na memória. Levando isso em consideração e também outros aspectos, pode-se destacar as seguintes vantagens e desvantagens:

Counting Sort:

A. Vantagens

- Velocidade Superior para Certos Cenários:O Counting Sort geralmente é mais rápido do que algoritmos de ordenação baseados em comparação, como merge sort e quicksort, especialmente quando o número de elementos (n) é grande e o valor máximo (k) é relativamente pequeno.
- Facilidade de Implementação: O Counting Sort é simples de codificar, tornando-o uma escolha acessível para desenvolvedores que precisam de uma solução rápida e eficiente para certas aplicações.
- Estabilidade: O Counting Sort é um algoritmo estável, o que significa que mantém a ordem relativa dos elementos iguais no array original.

B. Desvantagens

- Limitação a Inteiros Não Negativos: O Counting Sort funciona melhor com inteiros não negativos, limitando sua aplicabilidade para outros tipos de dados.
- Ineficiente para Intervalos Grandes: O Counting Sort se torna ineficiente se o intervalo de valores (k) a serem classificados for muito grande em relação ao número de elementos (n), devido ao aumento do uso de memória.
- Não é In-place: O Counting Sort não é um algoritmo de classificação in-place, pois requer espaço extra proporcional ao intervalo de valores para classificar os elementos da matriz.

Pigeonhole Sort

C. Vantagens

- Eficiência em Intervalos Pequenos: O Pigeonhole Sort é muito eficiente quando o intervalo de valores (k) no vetor de entrada é pequeno, pois evita muitas das comparações que outros algoritmos de classificação fazem.
- Facilidade de Implementação: O Pigeonhole Sort é fácil de entender e implementar, tornando-o uma escolha prática para problemas específicos.
- Estabilidade: O Pigeonhole Sort é um algoritmo estável, preservando a ordem relativa dos elementos iguais no vetor original.

D. Desvantagens

- Desempenho Dependente do Intervalo: Pode ser lento se o intervalo for muito maior que o número de elementos no vetor de entrada, levando a uma utilização ineficiente do tempo.
- Uso de Memória: O Pigeonhole Sort requer uma grande quantidade de memória para armazenar os pigeonholes, o que pode ser problemático se o intervalo de valores for muito grande.
- Não é In-place: O Pigeonhole Sort não é um algoritmo de classificação in-place, pois utiliza espaço extra para classificar os elementos do vetor.

IV. Modelos de Aplicação

Ambos os algoritmos trabalham de forma semelhante, porém possuem suas particularidades, sendo necessária a análise de alguns importantes tópicos no que tange à aplicação deles, como aplicação, tempo de execução, simplicidade de código, entre outros, os quais veremos a seguir.

A. Aplicações do Counting Sort

- Classificação de Notas de Exames: Em sistemas educacionais, o Counting Sort pode ser usado para classificar as notas de uma prova quando o intervalo de notas é restrito.
- Classificação de IDs: Em bancos de dados ou sistemas que gerenciam IDs únicos dentro de um intervalo pequeno, o Counting Sort pode acelerar a ordenação desses IDs.
- Análise de Frequências de Caracteres: Em processamento de texto, o Counting Sort pode ser utilizado para ordenar caracteres em um texto para análise de frequência, como em algoritmos de compressão de dados.

- Ordenação de Contagens em Análise Estatística: Em estatísticas e análise de dados, o Counting Sort pode ser usado para ordenar contagens de eventos ou ocorrências quando os dados são inteiros e o intervalo é limitado.
- Processamento de Imagens Digitais: Em alguns algoritmos de processamento de imagens, especialmente em operações que envolvem histograma, o Counting Sort pode ajudar a ordenar pixels por intensidade de cor.

B. Aplicações do Pigeonhole Sort

- Ordenação de Valores de Posição em Computação Gráfica: Na computação gráfica, os valores de posição dos pixels ou vértices podem estar dentro de um intervalo restrito, tornando o Pigeonhole Sort útil para ordenar rapidamente esses valores para operações de renderização ou transformações geométricas.
- Ordenação de Dados em Aplicações de Redes: Em algumas aplicações de redes, como roteamento e classificação de pacotes, os identificadores de pacotes ou endereços IP podem ser mapeados para intervalos restritos, permitindo a aplicação do Pigeonhole Sort para ordenar esses identificadores de maneira eficiente.
- Processamento de Dados em Tempo Real: Adequado para sistemas em tempo real que exigem ordenação rápida e eficiente de fluxos de dados com valores numéricos restritos. Exemplos incluem sistemas de monitoramento em tempo real e processamento de eventos.

V. Complexidade

Tanto o Counting Sort quanto o Pigeon Hole são algoritmos de ordenação lineares, e compartilham da mesma comlpexidade para todos os caos, sendo elas mostradas na tabela a seguir:

Melhor Caso	O(n+k)
Pior Caso	O(n+k)
Caso Médio	O(n+k)
Estabilidade	Sim

No Counting Sort, em todos os casos a complexidade é a mesma porque, independentemente da disposição dos elementos no array, o algoritmo percorre o array o mesmo número de vezes.

No Pigeonhole Sort, em todos os casos a complexidade também é a mesma, porque a criação dos buracos e a distribuição dos elementos são feitas independentemente da ordem inicial dos elementos de entrada.

VI. Resultados

Para visualizar os resultados dos dois algoritmos, ambos foram testados em 6 lingaugens diferentes lingagugens de Programação: C, C++, Java Script, Php, Java e Python, sendo o C e o C++ linguagens compiladas, o Java uma lingaugem intermidiária, isto é, comilada e interpretada, o Java Script e o Php linguagens interpretadas feitas para desenvolvimento web e o Python uma linguagem interpretada de alto nível. Os testes foram feitos com entradas (N) de diferentes tamanhos que variam de 0 a 100.000 elementos, a fim de se ter uma boa análise sobre como o Counting Sort e o Pigeonhole Sort se comportam em diferentes linguagens com N variando. Cada gráfico é o resultado da média de 10 compilações em cada linguagem. Além disso, deve ser considerado também a existência de k, já que ambos os algoritmos também dependem dele, sendo que no Counting Sort o k representa o elemento de valor máximo, e no Pigeonhole Sort o k representa o intervalo entre os elementos de valor de máximo e mínimo. Sabendo disso, foi estipulado que k vale 15.000 para a realização dos testes. Este valor foi tratado, então, como uma contante, o que não interfere na linearidade dos gráficos, pois, ainda que k deixasse de ser uma constante e tendesse à N, a complexidade passaria a ser 2N, o que também é linear.

Levando isso em consideração, foram gerados os seguintes gráficos:

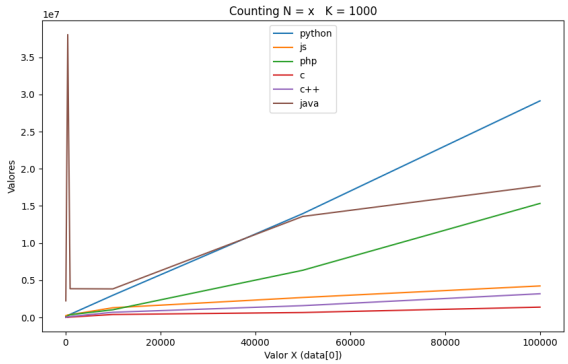

Fig. 12. Counting Sort n = x e k = 1000 gráfico

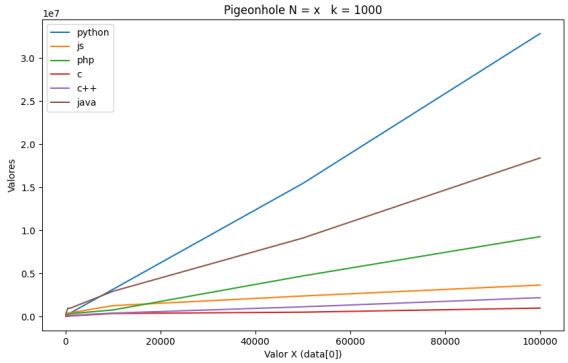

Fig. 13. Pigeonhole Sort n = x e k = 1000 gráfico

VII. Discussão

A partir dos gráficos expostos, pode-se fazer a seguinte análise:

A. Gráfico 1

- A curva inicial para Java apresenta um valor extremamente alto e, em seguida, estabiliza. Este comportamento não era esperado, e pode ser chamado ”dente” ou ”ruído”, representando alguma anomalia no comportamento da implementação. A razão disso pode ser diversas coisas, mas provavelmente está relacionada ou à atualizações de drivers da máquina ou às atualizações do Sistema Operacional da máquina.
- C e C++ apresentaram um tempo de execução muito baixo, e um custo compuatcional bom, o que se deve ao fato de serem linguagens compiladas de baixo nível, principalmente o C.
- Java Script teve um tempo de execução também baixo, próximo ao C++, o que se deve provavelmete às otimizações e tecnologias modernas utilizadas nos motores JavaScript, como o V8 do Google Chrome.
- Php teve um aumento no tempo de execução muito expressivo com relação ao C, C++ e Java Script, o que se deve à sua natureza interpetada, que gera uma sobrecarga maior, além do fato de que Php foi feito para desenvolvimento web, en não para tarefas que exigem processamento intendivo
- Python, por sua vez, teve o maior tempo de execução, uma vez que trata-se de uma lingaugem interpretada de alto nível, e por isso, tem de lidar com diversas operações para garantir a interpretação e simplicidade de sintaxe do código.

B. Gráfico 2

- De maneira geral, o Pigeonhole Sort exibe um comportamento linear para todas as linguagens, similar ao Counting Sort, mas sem a anomalia inicial observada no gráfico de Counting Sort para Java.
- C, C++ e Java Script continuam sendo as lingaugens que tiveram melhor desempenho, sendo o primeiro C, depois C++ e em seguida Jav Script.
- Php teve um tempo de execução menor do que teve com relação ao Counting Sort
- Java também apresentou uma melhora do tempo de execução.
- Python manteve-se com o maior tempo de execução, tendo, inclusive, este tempo auemntado em comparação ao Counting Sort.
  Tendo isso em vista, nota-se que o comportamento de algoritmos Counting Sorte e Pigeonhole Sort é semelhante, sendo o Pigeonhole Sort melhor, em termos de tempod e execução, em alguns casos, e o Counting Sort melhor, em outros.

VIII. Conclusão

Com base em tudo que foi apresentado, é possível perceber em que consistem ambos os algoritmos, como eles funcionam, se comportam e são aplicados, além de diversas outras características, como as vantagens e desvantagens, o que nos leva a uma conclusão acerca de tudo o que foi apresentado. Infere-se, portante, que o Counting Sort e Pigeonhole Sort são algoritmos de ordenação eficientes para conjuntos de dados onde a faixa de valores é limitada e relativamente pequena em comparação com o número de elementos.

Ambos os algoritmos são altamente eficientes em contextos específicos onde as condições de entrada são favoráveis. No entanto, para conjuntos de dados com uma ampla gama de valores ou em situações onde o uso de memória é uma preocupação, algoritmos de ordenação comparativos tradicionais, como Quick Sort ou Merge Sort, podem ser mais apropriados. A escolha entre Counting Sort e Pigeonhole Sort deve ser baseada nas características específicas dos dados a serem ordenados e nas restrições de recursos do sistema.

Conclui-se também que, em linguagens compiladas, como C e C++, os algoritmos têm um custo computacional mais favorável, enquanto em linguagens interpretadas, como Python, o desempenho pode decair devido à sobrecarga de interpretação.

Referências

[1] Livro ”Algoritmos Teoria e Prática” de Thomas H.Cormem, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein, 3° Edição.
[2] Site https://www.geeksforgeeks.org
[3] Livro ”Study and Comparison of Various Sorting Algorithms” de Ramesh Chand Pandey
[4] Site http://desenvolvendosoftware.com.br/algoritmos/ordenacao/counting-sort.html
[5] Site https://www.programiz.com/dsa/counting-sort