Implementação do Algoritmo de Ordenação Counting Sort para o Gerenciamento de Pedidos

Danilo Motta Rodriguesl[1], Gabriel da Silva Carvalho[2], Guilherme Viana Oliveira[3], Pedro Santos Simões[4] , Vitor Hugo Lima Silva Bittencourt[5]

[1]Sistema de informação – Centro Universitário de Excelência(Unex)

Daniloefectsafobe1234@gmail.com, bieuhcarvalho@gmail.com, guilherme13122106@gmail.com, pedrosimoes1298@gmail.com, vitorhugolima945@gmail.com

Abstract. *This report details the implementation of the Counting Sort algorithm in a food delivery order management system using Python. The algorithm was applied to efficiently sort data structures, specifically lists of dictionaries. The article discusses the fundamentals of Counting Sort as a sorting algorithm. The performance and stability of the implementation are proven in the context of system data management, such as sorting orders by their unique codes.*

Resumo. *Este relatório detalha a implementação do algoritmo Counting Sort em um sistema de gerenciamento de pedidos de food delivery usando Python. O algoritmo foi aplicado para ordenar eficientemente estruturas de dados, especificamente listas de dicionários. O artigo discute os fundamentos do Counting Sort como algoritmo de ordenação. A performance e a estabilidade da implementação são analisadas no contexto de gerenciamento de dados do sistema como a ordenação de pedidos por seus códigos únicos.*

1. Introducao

A ordenação de dados é uma operação fundamental na ciência da computação, essencial para a otimização de buscas, processamento e apresentação de informações de forma lógica e acessível. Em sistemas de software, como aplicações de gerenciamento de pedidos de food delivery, a capacidade de ordenar rapidamente grandes volumes de dados-sejam eles pedidos por ordem de chegada, itens por preço-impacta diretamente a eficiência operacional e a experiência do usuário. A escolha de um algoritmo de ordenação inadequado pode levar a gargalos de performance, especialmente à medida que a base de dados cresce.

O desafio de ordenação gerou uma variedade de algoritmos, cada um com suas próprias características de complexidade de tempo e espaço. Algoritmos baseados em comparação, com o Quick Sort ou Merge Sort, possuem um limite teórico. No entanto, para cenários específicos, como a ordenação de dados com chaves inteiras dentro de um intervalo conhecido, algoritmos de tempo linear, como o Counting Sort, oferecem uma alternativa significativamente mais rápida, Este relatório foca na implementação de Counting Sort, detalhando a aplicação prática para ordenar dados em um sistema Python refatorado.

2. Fundamentacao Teorica

O Counting Sort é um algoritmo de ordenação altamente eficiente, notável por operar em tempo linear. Sua principal característica é não ser baseado em comparações; ele não compara elementos entre si, como fazem o Quick Sort ou o Merge Sort. Em vez disso, ele determina a posição final de cada elemento no array ordenado com base em sua frequência e valor. Esta abordagem o torna ideal para ordenar coleções de dados cujas chaves são inteiros e estão dentro de um intervalo conhecido.

A estratégia do algoritmo é dividida em quatro etapas principais. Primeiro, o algoritmo identifica o valor máximo da chave (vamos chamá-la de K) no conjunto N elementos. Segundo, ele cria um array auxiliar de "contagem", geralmente de tamanho K+1, inicializado com zeros. Terceiro, o algoritmo itera sobre o array de entrada e usa o valor de cada elemento como um índice no array de contagem, incrementando a posição correspondente. Ao final desta etapa, o array de contagem armazena a frequência de cada elemento.

A quarta etapa é a mais crucial para a ordenação. O array de contagem é transformado em um array de soma cumulativa, onde cada posição i passa a armazenar a soma de todas as frequências até i. O valor em contagem[i] agora representa a posição final (mais um) do elemento i no array de saída. Finalmente, um array de saída é popular. O algoritmo itera pelo array de entrada (tipicamente de trás para frente, para garantir estabilidade) e, para cada elemento, consulta sua posição final no array de soma cumulativa, coloca o elemento no array de saída e decrementa o valor no array de soma cumulativa.

O Counting Sort possui características distintas. Sua complexidade de tempo é O(N + K), onde N é o número de elementos a serem ordenados e K é o valor máximo da chave. Se K for proporcional a N (ou seja, K = O(N), a complexidade torna-se O(N), ou tempo linear. No entanto, sua complexidade de espaço também é O(N + K), pois requer um array de saída de tamanho N é um array de contagem de tamanho K. Portanto, não é um algoritmo in-place. Uma de suas propriedades mais importantes é a estabilidade : elementos com chaves iguais mantêm sua ordem relativa original no array ordenado. Isso é garantido ao se iterar o array de entrada de forma reversa durante a construção do array de saída.

3. Metodologia

A implementação do projeto foi conduzida na linguagem Python, com foco em atender aos requisitos de refatoração e de implementação de algoritmos. O principal desafio metodológico foi adaptar o algoritmo Counting Sort, que classicamente opera sobre listas de inteiros, para ordenar uma lista de dicionários (dict), que é a estrutura de dados central do novo sistema.[1, 1] Para solucionar isso, o algoritmo foi implementado de forma generalizada.

Foi desenvolvida a função counting_sort_dicts(arr, key_name). Esta função recebe dois parâmetros: arr, que é a lista de dicionários a ser ordenada (ex: menu_de_itens ou todos_pedidos), e key_name, uma string que especifica qual chave dentro de cada

dicionário deve ser usada para a ordenação (ex: "id" ou "código"). A lógica central do Counting Sort foi preservada, incluindo a busca pelo valor máximo (k), a criação de um array de contagem baseado em frequência, e a transformação deste em um array de soma cumulativa para determinar as posições finais. A adaptação principal reside em como os valores são acessados (ex: d[key_name] em vez de apenas d) e na construção do array de saída, que é popular com os dicionários completos, garantindo uma ordenação estável.

Para atender ao requisito de ordenar os relatórios do sistema , esta função foi integrada em duas rotinas principais do código. Primeiro, na função consultar_itens(), o counting_sort_dicts é invocado para ordenar a lista menu_de_itens utilizando a chave "id", garantindo que os itens sejam exibidos em ordem crescente de seu código. Segundo, e mais crucial para a gestão de relatórios, a função exibir_pedidos() utiliza o counting_sort_dicts para ordenar a lista todos_pedidos pela chave "código", antes de apresentá-los ao usuário.

```python
def counting_sort_dicts(arr, key_name):
    if not arr:
        return []
    max_val = 0
    for d in arr:
        if d[key_name] > max_val:
            max_val = d[key_name]
    count = [0] * (max_val + 1)
    for d in arr:
        count[d[key_name]] += 1
    for i in range(1, len(count)):
        count[i] += count[i-1]
    output = [None] * len(arr)
    for d in reversed(arr):
        element_val = d[key_name]
        output_index = count[element_val] - 1
        output[output_index] = d
        count[element_val] -= 1
    return output

def consultar_itens():
    if not menu_de_itens:
        print("\n⚠ Nenhum item cadastrado.\n")
        return
    print("\n Lista de Itens (Ordenada por ID):")
    lista_ordenada = counting_sort_dicts(menu_de_itens, "id")
    for item in lista_ordenada:
        print(f"[{item["id"]}] {item["nome"]} - R${item["preco"]:.2f} (Estoque: {item["estoque"]})")
    print()
```

4. Resultados e Discussões

Os resultados práticos obtidos com a implementação do algoritmo Counting Sort, conforme descrito na metodologia. O objetivo é demonstrar o comportamento do algoritmo na ordenação dos dados do sistema (menu_de_itens e todos_pedidos) e discutir a eficácia da solução adotada.

```
menu_de_itens = []
todos_pedidos = []
```

Os pedidos são listados em ordem numérica crescente de seus códigos, confirmando o funcionamento correto da implementação. A adaptação do algoritmo para operar sobre dicionários, utilizando o parâmetro key_name ("id" ou "código"), mostrou-se eficaz. Além disso, a implementação manteve a estabilidade do algoritmo , garantida pela iteração reversa (reversed(arr)) durante a construção do array de saída. Isso significa que se houvesse duas chaves de ordenação idênticas, a ordem relativa original delas seria preservada.

Para detalhar visualmente a lógica implementada na função counting_sort_dicts, a imagem abaixo apresenta um fluxograma que ilustra as etapas principais: (1) encontrar o valor máximo (k), (2) criar e popular o array de contagem, (3) calcular a soma cumulativa para encontrar as posições, e (4) construir o array de saída ordenado.

```
def counting_sort_dicts(arr, key_name):
    if not arr:
        return []
    max_val = 0
    for d in arr:
        if d[key_name] > max_val:
            max_val = d[key_name]
    count = [0] * (max_val + 1)
    for d in arr:
        count[d[key_name]] += 1
    for i in range(1, len(count)):
        count[i] += count[i-1]
    output = [None] * len(arr)
    for d in reversed(arr):
        element_val = d[key_name]
        output_index = count[element_val] - 1
        output[output_index] = d
        count[element_val] -= 1
    return output
```

```
def exibir_pedidos():
    print("\n--- LISTA DE PEDIDOS (Ordenada) ---")
    lista_pedidos_ordenados = counting_sort_dicts(todos_pedidos, "codigo")
    for pedido in lista_pedidos_ordenados:
        print(f"Código: {pedido['codigo']} | Cliente: {pedido['nome_cliente']} | Valor: {pedido['valor_total']:.2f} |Status: {pedido['status']}")
    print("-------------------------\n")
```

Em termos de complexidade e desempenho, a discussão foca na complexidade de tempo do Counting Sort, que é O(N + k), onde N é o número de elementos (pedidos) e K é o valor máximo da chave (o maior código de pedido). Em cenários onde K é proporcional a N, o desempenho é linear, superando algoritmos baseados em comparação O(N log N). No entanto, isso introduz uma discussão relevante: a

complexidade de espaço é O(N + K) devido ao array de contagem. Se o valor K (ex: o ID de um pedido) se tornar desproporcionalmente grande (ex: um sistema que usa códigos de 8 dígitos), o consumo de memória para o array de contagem pode se tornar um gargalo, sendo um ponto de atenção para a escalabilidade da solução.

5. Considerações Finais

Este relatório detalhou a análise e a implementação do algoritmo de ordenação Counting Sort , como parte do desafio de refatoração de um sistema de gerenciamento de pedidos. O projeto atingiu seu objetivo principal ao implementar com sucesso a função counting_sort_dicts e integrá-la ao sistema para ordenar relatórios de itens e pedidos por suas respectivas chaves numéricas ("id" e "código") , atendendo aos requisitos estipulados.

Entre os pontos desafiadores , destaca-se a necessidade de generalizar o algoritmo. O Counting Sort clássico opera sobre inteiros; adaptá-lo para ordenar uma lista de dicionários Python com base em uma key_name exigiu uma atenção cuidadosa ao acesso dos dados e à construção do array de saída. Além disso, garantir a estabilidade do algoritmo (mantendo a ordem relativa de pedidos com chaves idênticas) foi um ponto crucial de implementação, solucionado com a iteração inversa durante a alocação dos elementos no array de saída.

Como pontos de interesse , a implementação prática de um algoritmo de tempo linear O(N + K) forneceu uma visão clara de suas vantagens de desempenho sobre algoritmos baseados em comparação O(N log N) para conjuntos de dados apropriados. A discussão sobre a complexidade de espaço O(N + K) também foi relevante, identificando o consumo de memória como a principal limitação da abordagem caso o valor máximo da chave K se torne astronomicamente grande.

6. Referencias

https://www.geeksforgeeks.org/dsa/counting-sort

https://www.programiz.com/dsa/counting-sort

https://docs.python.org/3/

https://docs.python.org/pt-br/3/library/json.html