CURSO: CIÊNCIA DA COMPUTAÇÃO

DISCIPLINA: VELOCIDADE E QUALIDADE COM ESTRUTURAS DE DADOS E ALGORITMOS

ANÁLISE DE UM ALGORITMO DE ORDENAÇÃO MANUAL RESTRITO:

UM ESTUDO SOBRE A COMPLEXIDADE O(N²) IMPULSIONADA POR UM MODELO DE ACESSO A DADOS RESTRITO

Fábio Linhares[1]

RESUMO: Este artigo apresenta uma análise formal de um algoritmo de ordenação de duas fases, concebido para operar sob severas restrições físicas análogas à ordenação de um baralho de cartas numa única mão. Modelamos estas restrições utilizando estruturas de dados abstratas — especificamente, uma pilha de origem e um número fixo de deques auxiliares — para formalizar a lógica operacional do algoritmo. Através de uma análise assintótica rigorosa, demonstramos que o algoritmo exibe uma complexidade de tempo de pior caso e caso médio de e uma complexidade de tempo de melhor caso de Theta(n). Estabelecemos que este desempenho quadrático não é um artefacto de implementação, mas uma consequência intrínseca do modelo de acesso a dados restrito, que impede o uso de paradigmas mais eficientes de O(nlogn). Ao traçar paralelos diretos com algoritmos clássicos como o Insertion Sort e ao apresentar uma implementação funcional em Python, contextualizamos o desempenho do algoritmo e provamos que ele representa uma solução correta, embora computacionalmente ineficiente, ditada pelo seu

[1] Mestrando em Informática na Universidade Federal de Alagoas (UFAL).
Especialista em Inteligência Artificial e Aprendizado de Máquina (PUC Minas).
Graduado em Banco de Dados, BI e Big Data (Infnet)
Monitor no Instituto Infnet.

ambiente operacional único.

1 INTRODUÇÃO

1.1 DEFINIÇÃO DO PROBLEMA

O problema em questão envolve a ordenação de um subconjunto de um baralho de cartas padrão — especificamente, as 13 cartas do naipe de espadas, de Ás (1) a Rei (13). O desafio não reside na tarefa de ordenação em si, mas nas rigorosas restrições físicas impostas ao processo, que simulam a manipulação de cartas com um número limitado de mãos e movimentos. O objetivo deste trabalho é analisar a eficiência computacional do algoritmo que emerge naturalmente destas limitações.

As restrições físicas, que formam a tese central desta análise, são as seguintes:

1. **Fonte Única com Acesso Sequencial:** A pilha inicial de cartas embaralhadas só permite que a carta do topo seja vista e removida. Qualquer carta abaixo dela só se torna acessível após a remoção de todas as cartas que a precedem. Este modelo de acesso é funcionalmente idêntico à propriedade fundamental de uma estrutura de dados de pilha (stack) Last-In, First-Out (LIFO).[1]
2. **Armazenamento Intermediário Limitado:** As cartas são ordenadas em um número limitado de pilhas intermediárias, metaforicamente "seguras entre os dedos". Este limite corresponde a um número pequeno e constante de estruturas de dados auxiliares disponíveis para o algoritmo.
3. **Pontos de Inserção Restritos:** Uma carta só pode ser colocada no início ou no fim de uma pilha intermediária existente. A inserção direta no meio de uma pilha é proibida. Esta é a característica definidora de uma fila de duas pontas, ou deque (double-ended queue).[2]
4. **Transferência de Elemento Unitário:** Apenas uma carta pode ser movida de cada vez. Esta regra proíbe operações em subconjuntos ou subsequências de cartas, uma capacidade que é fundamental para muitos algoritmos de ordenação avançados.

1.2 OBJETIVOS DA PESQUISA E ESTRUTURA DO ARTIGO

O objetivo principal deste artigo é conduzir uma análise formal e rigorosa da complexidade computacional do algoritmo de ordenação manual derivado das restrições acima mencionadas.[5] A hipótese central é que a complexidade quadrática observada no algoritmo não é uma falha de projeto, mas sim um resultado inevitável e matematicamente demonstrável do seu modelo de acesso a dados inerentemente restrito.

Para atingir este objetivo, o artigo está estruturado da seguinte forma:

- A **Secção 2.0** traduz as restrições físicas para um modelo formal utilizando estruturas de dados abstratas, redefinindo o problema em termos da ciência da computação.
- A **Secção 3.0** fornece uma descrição formal e agnóstica de linguagem do algoritmo de duas fases que emerge naturalmente para resolver o problema sob as restrições dadas.
- A **Secção 4.0** constitui o núcleo analítico do trabalho, apresentando provas rigorosas para os limites de complexidade de tempo (pior, médio e melhor caso) e de espaço do algoritmo.
- A **Secção 5.0** contextualiza o algoritmo dentro da teoria da ordenação estabelecida, comparando-o com algoritmos canónicos e justificando a inaplicabilidade de paradigmas mais eficientes.
- Finalmente, a **Secção 6.0** resume as conclusões e reitera o princípio fundamental de que as restrições da estrutura de dados ditam diretamente a complexidade algorítmica.

2 FORMALIZANDO AS RESTRIÇÕES: UM MODELO DE ESTRUTURA DE DADOS ABSTRATA

Para analisar o processo de ordenação manual com rigor matemático, é essencial traduzir as ações físicas e as suas limitações para um modelo abstrato. Este ato de formalização permite-nos aplicar as ferramentas da análise de algoritmos e da teoria da complexidade. O problema, que à primeira vista parece ser sobre destreza manual, revela-se ser um problema bem definido no domínio da ordenação com estruturas de dados de acesso restrito.

2.1 A PILHA DE ORIGEM COMO UMA PILHA (STACK)

A pilha inicial de cartas embaralhadas, onde apenas a carta do topo é visível e acessível, é modelada de forma precisa por uma estrutura de dados de pilha (stack). As operações permitidas no mundo físico têm correspondentes diretos nas operações de pilha [1]:

- **Ver a carta do topo:** Corresponde à operação peek() ou top(), que retorna o elemento no topo da pilha sem o remover.
- Retirar a carta do topo: Corresponde à operação pop(), que remove e retorna o elemento do topo.

  A natureza LIFO (Last-In, First-Out) da pilha captura perfeitamente a restrição de acesso sequencial, onde a ordem de remoção é o inverso da ordem de empilhamento.

2.2 PILHAS INTERMEDIÁRIAS COMO DEQUES

As pilhas de cartas mantidas "entre os dedos" servem como armazenamento de trabalho. A restrição crucial aqui é que uma nova carta só pode ser adicionada no topo (frente) ou na base (trás) de uma destas pilhas. Esta funcionalidade é a definição exata de uma fila de duas pontas, ou deque.[4]

- **Colocar uma carta no topo:** Corresponde à operação insert_front().
- **Colocar uma carta na base:** Corresponde à operação insert_rear().

É fundamental notar que, com uma implementação subjacente eficiente (como uma lista duplamente ligada), ambas as operações insert_front() e insert_rear(), bem como as operações de remoção correspondentes (delete_front(), delete_rear()), têm uma complexidade de tempo de O(1).[2] Esta distinção é vital. A análise preliminar do nosso algoritmo inicial (Resolução de Exercícios - Estruturas de Dados e Algoritmos, exercício 2) destaca corretamente que a operação list.insert(0,...) em Python tem um custo de O(n), mas isso é um artefacto da implementação de listas como arrays dinâmicos. Para uma análise teórica pura do algoritmo, assumimos agora a estrutura de dados mais apropriada (deque) e os seus custos operacionais ideais de O(1). Isto permite-nos isolar a complexidade intrínseca do algoritmo daquela complexidade induzida pela escolha da ferramenta de implementação.

2.3 O PROBLEMA DE ORDENAÇÃO REDEFINIDO

Com este modelo formal, o problema de ordenação manual pode ser redefinido em termos precisos da ciência da computação:

Dada uma pilha de origem S contendo n elementos não ordenados, e um número constante k de deques auxiliares D1,D2,...,Dk, desenvolver um algoritmo para ordenar os elementos numa única sequência ordenada, utilizando apenas as operações permitidas de pilha e deque.

Esta reformulação é o passo crítico que desbloqueia a capacidade de aplicar uma análise assintótica rigorosa e de situar o problema no vasto corpo da literatura sobre algoritmos de ordenação.

3 O ALGORITMO DE ORDENAÇÃO DE DUAS FASES: DESCRIÇÃO FORMAL

O algoritmo que emerge para resolver o problema redefinido, conforme descrito no documento fornecido [5], opera em duas fases distintas: uma fase de **Distribuição**, onde os elementos são movidos da pilha de origem para os deques auxiliares, e uma fase de **Consolidação**, onde os elementos nos deques são fundidos numa única sequência ordenada. A seguir, apresentamos uma

descrição formal e agnóstica de linguagem deste processo.

3.1 FASE I: DISTRIBUIÇÃO

Nesta fase, cada elemento é retirado da pilha de origem e inserido num dos deques auxiliares, seguindo as regras de inserção restritas. Se uma inserção direta não for possível, o algoritmo tenta reorganizar os elementos entre os deques para criar uma oportunidade de inserção. Na prática, é assim:

Para cada carta retirada do topo:

1. Tenta-se inserir a carta de forma ordenada:
   - Se uma pilha está vazia, inicia-se essa pilha com a carta.
   - Se a carta é menor que o primeiro elemento de uma pilha, insere-se no início.
   - Se é maior que o último elemento, insere-se no fim.
2. Se a carta não coube, realizam-se movimentos auxiliares entre as pilhas para abrir espaço e tenta-se novamente a inserção.

Pseudocódigo:

```
Algoritmo: Distribuir(Pilha S, Deques D[1..k])
  enquanto S não está vazia:
    elemento ← S.pop()
    colocado ← falso

    // Tenta a inserção direta
    para i de 1 até k:
      se D[i] está vazio:
        D[i].insert_rear(elemento)
        colocado ← verdadeiro
        quebrar
      senão se elemento < D[i].get_front():
        D[i].insert_front(elemento)
        colocado ← verdadeiro
        quebrar
      senão se elemento > D[i].get_rear():
        D[i].insert_rear(elemento)
        colocado ← verdadeiro
        quebrar
    fim para

    // Se a inserção direta falhou, tenta abrir espaço
    se não colocado:
      Abrir_Espaço_E_Inserir(elemento, D)
  fim enquanto
```

O subprocedimento Abrir_Espaço_E_Inserir encapsula a lógica de movimentação de elementos entre os deques D1,...,Dk. Esta é uma estratégia gulosa que procura um movimento válido (retirar um elemento da frente de um deque e inseri-lo na frente ou no fim de outro) que permita a inserção subsequente do elemento atual. A sua implementação exata pode variar, mas o seu objetivo é resolver impasses de inserção através de reorganização interna.

3.2 FASE II: CONSOLIDAÇÃO

Após todos os elementos terem sido distribuídos, esta fase visa fundir os deques, que contêm sequências parcialmente ordenadas (corridas), numa única deque totalmente ordenada.

Pseudocódigo:

```
Algoritmo: Consolidar(Deques D[1..k])
  enquanto mais de um deque em D não está vazio:
    // Encontra o melhor movimento inter-deques possível que combine corridas
    (origem, destino, elemento, posição) ← Encontrar_Melhor_Movimento(D)

    // Executa o movimento
    D[origem].pop_front() // Assumindo que os movimentos são sempre da frente
    se posição == 'frente':
      D[destino].insert_front(elemento)
    senão: // posição == 'trás'
      D[destino].insert_rear(elemento)
    fim se
  fim enquanto

  retornar o único deque não vazio
```

A função Encontrar_Melhor_Movimento examina as frentes e os fins de todos os deques não vazios para encontrar um movimento que funda duas corridas de forma válida (por exemplo, mover o elemento da frente de Di para a frente de Dj se for menor que o elemento da frente de Dj). Este processo é repetido até que todos os elementos residam numa única estrutura.

4 ANÁLISE DE COMPLEXIDADE RIGOROSA

Esta secção apresenta a análise formal da complexidade do algoritmo, fundamentando as suas características de desempenho em princípios matemáticos. A análise foca-se no número de operações primitivas executadas em função do número de elementos, n.

4.1 CUSTO DAS OPERAÇÕES PRIMITIVAS

Como estabelecido na Secção 2, assumimos um modelo de dados onde as estruturas de pilha e deque são implementadas de forma eficiente. Portanto, todas as operações fundamentais — push(), pop(), insert_front(), insert_rear(), delete_front(), delete_rear(), get_front(), e get_rear() — são consideradas operações de tempo constante, ou seja, O(1).[1] A complexidade do algoritmo global é, portanto, determinada pelo número total de vezes que estas operações de O(1) são invocadas.

4.2 ANÁLISE DA COMPLEXIDADE DE TEMPO NO PIOR CASO (Θ(N2))

O pior caso para um algoritmo de ordenação ocorre frequentemente quando a entrada está ordenada na ordem inversa da desejada. Para uma ordenação ascendente, uma sequência de entrada como [13,12,11,...,1] serve como um exemplo canónico.

Prova: Consideremos o processo de inserção de n elementos em ordem decrescente numa estrutura que deve ser mantida ordenada ascendentemente. A lógica do algoritmo é análoga à do Insertion Sort ou à ordenação de uma pilha com uma pilha auxiliar.[6]

1. O primeiro elemento, n, é inserido. Custo: O(1).
2. O segundo elemento, n−1, é retirado da pilha de origem. Para o inserir na posição correta (antes de n), pode ser necessário mover n para um deque temporário, inserir n−1, e depois mover n de volta. No modelo do algoritmo, isto manifesta-se como uma inserção na frente. Custo: O(1).
3. O terceiro elemento, n−2, é retirado. Para o inserir antes de n−1, o mesmo processo ocorre.

O verdadeiro custo quadrático torna-se evidente durante a consolidação ou quando o subprocedimento Abrir_Espaço_E_Inserir é invocado repetidamente. Imagine que, para inserir o i-ésimo elemento, o algoritmo precisa de realizar uma série de movimentos internos para criar espaço. No pior cenário, para cada novo elemento, pode ser necessário deslocar uma quantidade de elementos proporcional ao número de elementos já processados (i−1).

Este comportamento é mais claramente visto no problema análogo de ordenar uma pilha com uma pilha auxiliar.[6] Para inserir o i-ésimo elemento (em ordem inversa) na pilha ordenada auxiliar, é necessário:

- pop() de i−1 elementos da pilha auxiliar para a pilha de origem.
- push() do i-ésimo elemento na pilha auxiliar.
- push() dos i−1 elementos de volta da pilha de origem para a auxiliar.

O número total de operações para o i-ésimo elemento é proporcional a i. Somando para todos os

n elementos, o número total de operações T(n) é dado pela soma de uma progressão aritmética:

$$T(n)=i=1\sum nc\cdot i=c\cdot 2n(n+1)$$

onde c é uma constante. Esta soma simplifica para $c\cdot(21n2+21n)$, que tem uma complexidade assintótica de $\Theta(n2)$.8 A análise formal do Insertion Sort chega à mesma conclusão para o seu pior caso.10

4.3 ANÁLISE DA COMPLEXIDADE DE TEMPO NO MELHOR CASO (Θ(N))

O melhor caso ocorre quando a entrada já está ordenada (ou quase ordenada). Para uma ordenação ascendente, uma sequência de entrada como [1,2,3,...,13].

Prova:

1. O primeiro elemento, 1, é retirado da pilha de origem e colocado na deque D1. Custo: O(1).
2. O segundo elemento, 2, é retirado. Como 2 é maior que o último elemento em D1 (que é 1), ele é simplesmente adicionado ao fim de D1 com uma operação insert_rear(). Custo: O(1).
3. Este padrão continua para todos os n elementos. Cada elemento requer uma operação pop() da pilha de origem e uma operação insert_rear() num deque, ambas de custo O(1).

A fase de distribuição completa-se em $n\times O(1)=O(n)$ operações. A fase de consolidação é trivial, pois todos os elementos já se encontram num único deque ordenado, não requerendo movimentos adicionais. Portanto, a complexidade de tempo total no melhor caso é $\Theta(n)$. Este resultado é consistente com o melhor caso do Insertion Sort, que também é linear quando a entrada já está ordenada.[11]

4.4 ANÁLISE DA COMPLEXIDADE DE TEMPO NO CASO MÉDIO (Θ(N2))

Para uma entrada com uma permutação aleatória de elementos, o desempenho do algoritmo tende para o do pior caso. A lógica fundamental do algoritmo é pegar um elemento de cada vez e inseri-lo na sua posição correta dentro de uma estrutura crescente de elementos já ordenados (ou parcialmente ordenados). Esta é a definição operacional do Insertion Sort.[11]

Para uma entrada aleatória, espera-se que um novo elemento, ao ser inserido numa subsequência já ordenada de tamanho i, precise de ser comparado com, em média, i/2 elementos para encontrar a sua posição.[8] No nosso modelo, estas "comparações" e "deslocamentos" são implementados através de uma série de operações de pop e push entre os deques. O número de tais operações para inserir o i-ésimo elemento ainda é, em média, proporcional a i. A complexidade de tempo

no caso médio é, portanto, a soma dos custos médios para cada inserção:

$$T(n) = \sum_{i=1}^{n} c \cdot 2i = 2c \cdot 2n(n+1)$$

Esta expressão também resulta numa complexidade de $\Theta(n2)$. Assim, tal como o Insertion Sort e o Selection Sort, o desempenho médio do algoritmo é quadrático.8 O algoritmo não é adaptativo no sentido de tirar proveito de uma ordem parcial, exceto no caso totalmente ordenado.16

4.5 ANÁLISE DA COMPLEXIDADE DE ESPAÇO (Θ(N))

A complexidade de espaço refere-se à memória adicional necessária para além da que armazena a entrada inicial. No modelo deste algoritmo, todos os n elementos da pilha de origem são eventualmente transferidos para os deques auxiliares. No pico do uso de memória, todos os n elementos residem nestas estruturas auxiliares.

Portanto, o espaço auxiliar necessário é diretamente proporcional ao número de elementos, n. A complexidade de espaço do algoritmo é $\Theta(n)$. Isto contrasta com algoritmos de ordenação "in-place" (no local) como o Heapsort ou a implementação padrão do Insertion Sort, que requerem apenas uma quantidade constante de espaço auxiliar, $O(1)$.[13]

5.0 ANÁLISE COMPARATIVA E CONTEXTO TEÓRICO

Para compreender plenamente o desempenho do algoritmo, é essencial situá-lo no contexto mais amplo da teoria dos algoritmos de ordenação. Esta secção compara o algoritmo com paradigmas estabelecidos e explica por que razão as suas restrições o confinam a uma classe de desempenho quadrática.

5.1 COMPARAÇÃO DIRETA COM ORDENAÇÃO BASEADA EM PILHAS

O problema em análise é uma variante de um problema clássico da ciência da computação: "ordenar uma pilha utilizando apenas uma pilha auxiliar". Este problema é um exercício padrão para ilustrar a recursão e a manipulação de pilhas, e a sua solução ótima tem uma complexidade de tempo de $\Theta(n2)$.[6]

A lógica da solução canónica envolve retirar um elemento da pilha de origem e, para o inserir na sua posição correta na pilha auxiliar (que é mantida ordenada), mover elementos maiores da pilha auxiliar de volta para a de origem. Este processo de "fazer espaço" para cada um dos n

elementos pode exigir até O(n) movimentos, resultando no comportamento quadrático geral.

O algoritmo aqui analisado utiliza múltiplos deques em vez de uma única pilha auxiliar. Esta abordagem oferece mais flexibilidade — existem 2k pontos de inserção (a frente e o fim de cada um dos k deques) em vez de apenas um. No entanto, esta flexibilidade apenas proporciona uma melhoria por um fator constante. Não altera a natureza fundamental do problema: a ausência de acesso aleatório continua a exigir uma busca linear (implementada por movimentos de elementos) para encontrar o ponto de inserção correto. A complexidade assintótica permanece, portanto, Θ(n2).

5.2 ANALOGIAS COM ALGORITMOS DE ORDENAÇÃO QUADRÁTICOS CANÓNICOS

INSERTION SORT

A analogia mais forte e direta é com o **Insertion Sort** (Ordenação por Inserção).[11] O Insertion Sort constrói a matriz ordenada final um elemento de cada vez. Para cada novo elemento, ele percorre a sub-matriz já ordenada para encontrar a posição correta, deslocando os elementos maiores para abrir espaço.

O algoritmo manual é uma re-implementação da lógica processual do Insertion Sort num modelo de dados não-aleatório.

- O "elemento a ser inserido" é a carta retirada da pilha de origem.
- A "sub-matriz ordenada" são as corridas ordenadas mantidas nos deques.
- A operação de "percorrer e deslocar" do Insertion Sort é substituída pelas fases de Abrir_Espaço_E_Inserir e Consolidar, que utilizam uma série de operações pop e push de O(1) para atingir o mesmo resultado lógico.

Como a estrutura algorítmica de alto nível — inserir iterativamente cada um dos n elementos numa estrutura ordenada crescente de tamanho até n−1 — é a mesma, a complexidade de tempo também é conservada.

SELECTION SORT

Uma analogia mais fraca pode ser feita com o **Selection Sort** (Ordenação por Seleção).[17] Uma passagem do Selection Sort envolve percorrer toda a porção não ordenada da lista para encontrar o elemento mínimo e trocá-lo para a posição correta. A fase de

Consolidar do nosso algoritmo, ao tentar encontrar o próximo elemento mais pequeno entre as

frentes de todos os deques para o mover para a deque final, realiza uma operação conceptualmente semelhante de "procurar o mínimo". No entanto, a natureza incremental e de construção um-a-um do processo global alinha-se muito mais de perto com o Insertion Sort.

5.3 INAPLICABILIDADE DE ALGORITMOS O(NLOGN)

Uma parte crucial da análise é demonstrar que algoritmos mais eficientes, com complexidade O(nlogn), são fundamentalmente incompatíveis com as restrições impostas. Isto prova que o desempenho Θ(n2) não é uma escolha, mas uma necessidade.

- **Quicksort:** A eficiência do Quicksort depende criticamente da sua operação partition, que reorganiza uma sub-matriz em torno de um pivô. Esta operação requer acesso aleatório para comparar e trocar eficientemente elementos de diferentes partes da sub-matriz, muitas vezes das extremidades opostas em direção ao centro.[20] Numa pilha ou deque, aceder a um elemento do meio para servir de pivô, ou trocar dois elementos não adjacentes, exigiria O(n) operações. Isto degradaria a operação de partição para O(n2) e o algoritmo Quicksort global para, pelo menos, O(n3).
- **Mergesort:** O Mergesort baseia-se na estratégia de "dividir para conquistar". A sua fase de "dividir" requer a capacidade de dividir eficientemente uma coleção em duas metades. Num array, isto é uma operação O(1) (simplesmente calculando um índice do meio). Numa pilha, dividir a coleção em duas sub-pilhas de tamanho n/2 exigiria a movimentação de n/2 elementos, uma operação de O(n).[22] Embora a fase de "conquistar" (merge), que combina duas listas ordenadas, seja bem adequada ao acesso sequencial, a ineficiência da fase de divisão torna uma implementação direta do Mergesort impraticável e ineficiente sob estas restrições.

5.4 TABELA COMPARATIVA

A tabela seguinte resume as propriedades do algoritmo de ordenação manual restrito em comparação com algoritmos de ordenação canónicos, destacando como o seu perfil de desempenho se alinha com os algoritmos quadráticos e por que se distingue dos algoritmos mais eficientes.

Algoritmo	Tempo Melhor Caso	Tempo Caso Médio	Tempo Pior Caso	Espaço Auxiliar	Modelo de Acesso a Dados	Estável

Ordenação Manual Restrita	Θ(n)	Θ(n2)	Θ(n2)	Θ(n)	Pilha / Deque	Sim
Insertion Sort	Θ(n)	Θ(n2)	Θ(n2)	O(1)	Acesso Aleatório	Sim
Selection Sort	Θ(n2)	Θ(n2)	Θ(n2)	O(1)	Acesso Aleatório	Não
Mergesort	Θ(nlogn)	Θ(nlogn)	Θ(nlogn)	O(n)	Sequencial	Sim
Quicksort	Θ(nlogn)	Θ(nlogn)	Θ(n2)	O(logn)	Acesso Aleatório	Não
Ordenação de Pilha (1 aux)	Θ(n2)	Θ(n2)	Θ(n2)	Θ(n)	Pilha	Sim

Nota sobre Estabilidade: Um algoritmo de ordenação é estável se não alterar a ordem relativa de elementos com valores iguais. O algoritmo manual restrito é estável porque os novos elementos são sempre colocados estritamente antes ou depois de corridas existentes, nunca reordenando elementos de valor igual dentro de uma corrida.[16]

6.0 CONCLUSÃO

6.1 RESUMO DOS RESULTADOS

Este artigo realizou uma análise formal do algoritmo de ordenação manual apresentado aqui: (Resolução de Exercícios - Estruturas de Dados e Algoritmos, Exercício 2) dado o conjunto único e severo de restrições operacionais. Ao modelar o problema com estruturas de dados abstratas — uma pilha de origem e um conjunto de deques auxiliares — foi possível aplicar uma análise de complexidade rigorosa. Os resultados confirmam que o algoritmo de duas fases (Distribuição e Consolidação) possui os seguintes limites de desempenho:

- **Complexidade de Tempo:** Θ(n) no melhor caso (entrada ordenada), e Θ(n2) no caso médio e no pior caso (entrada aleatória ou inversamente ordenada).
- **Complexidade de Espaço:** Θ(n) de espaço auxiliar, uma vez que todos os elementos devem ser armazenados nas estruturas de trabalho.

6.2 A PRIMAZIA DAS RESTRIÇÕES

A conclusão mais significativa desta análise é que o desempenho quadrático do algoritmo não é um indicativo de um projeto subótimo, mas sim uma consequência inerente e inevitável das restrições de acesso a dados impostas pelo problema. A incapacidade de realizar acesso aleatório, de particionar eficientemente a coleção ou de operar em subconjuntos de dados impede a aplicação de paradigmas de "dividir para conquistar" que sustentam algoritmos de ordenação com complexidade O(nlogn). O modelo de acesso restrito força uma abordagem iterativa e linear para a inserção de cada elemento, um padrão que é fundamentalmente quadrático.

6.3 AVALIAÇÃO FINAL

Dado o enquadramento único do problema, nosso algoritmo se mostra como uma solução lógica, funcionalmente correta e, em certo sentido, ótima dentro do seu universo de operações permitidas. Ele serve como um caso de estudo exemplar que ilustra um dos princípios mais fundamentais da ciência da computação: a estrutura de dados e o seu modelo de acesso não são meros detalhes de implementação, mas sim forças primárias que ditam a complexidade e a eficiência dos algoritmos que podem operar sobre eles. **O algoritmo transforma uma tarefa manual aparentemente simples num exemplo prático e tangível da teoria da complexidade computacional.**

APENDICE

A implementação completa do algoritmo em Python, que encapsula a lógica descrita no artigo:.

```
import collections

def encontrar_lugar_para_inserir(numero, listas):
    """
    Tenta encontrar um lugar para o número nas listas auxiliares.
    Retorna a chave da lista e a posição ('inicio' ou 'fim').
    """
    # Prioriza listas vazias
    for nome_lista, lista in listas.items():
        if not lista:
            return nome_lista, 'fim'

    # Verifica se pode inserir no início ou no fim
    for nome_lista, lista in listas.items():
        if numero < lista[0]:
```

```
            return nome_lista, 'inicio'
        if numero > lista[-1]:
            return nome_lista, 'fim'

    return None, None

def mover_entre_listas_para_abrir_espaco(listas, historico_movimentos):
    """
    Move um elemento entre as listas para abrir espaço, seguindo regras
heurísticas.
    Retorna True se um movimento foi feito, False caso contrário.
    """
    listas_ativas = {k: v for k, v in listas.items() if v}
    if len(listas_ativas) < 2:
        return False

    nome_destino = max(listas_ativas, key=lambda k: listas_ativas[k][0])
    lista_destino = listas_ativas[nome_destino]

    candidatos_a_mover = []
    for nome_origem, lista_origem in listas_ativas.items():
        if nome_origem == nome_destino:
            continue

        num_a_mover = lista_origem[0]
        movimento_proposto = (nome_origem, nome_destino, num_a_mover)

        if num_a_mover < lista_destino[0] and (nome_destino, nome_origem,
num_a_mover) not in historico_movimentos:
            candidatos_a_mover.append({'num': num_a_mover, 'origem':
nome_origem, 'pos': 'inicio'})
        elif num_a_mover > lista_destino[-1] and (nome_destino, nome_origem,
num_a_mover) not in historico_movimentos:
             candidatos_a_mover.append({'num': num_a_mover, 'origem':
nome_origem, 'pos': 'fim'})

    if not candidatos_a_mover:
        return False

    primeiro_do_destino = lista_destino[0]
    melhor_candidato = min(candidatos_a_mover, key=lambda c: abs(c['num'] -
primeiro_do_destino))

    num_movido = listas[melhor_candidato['origem']].pop(0)
    if melhor_candidato['pos'] == 'inicio':
        listas[nome_destino].insert(0, num_movido)
    else:
        listas[nome_destino].append(num_movido)

    historico_movimentos.add((melhor_candidato['origem'], nome_destino,
num_movido))

    return True
```

```
def consolidar_listas(listas):
    """
    Fase final: move todos os elementos para uma única lista ordenada.
    """
    historico_movimentos = set()
    while sum(1 for lista in listas.values() if lista) > 1:
        movimento_feito = mover_entre_listas_para_abrir_espaco(listas,
historico_movimentos)
        if not movimento_feito:
            break

    for lista in listas.values():
        if lista:
            return lista
    return []

# --- ALGORITMO PRINCIPAL ---
def ordenar_cartas(A):
    listas_trabalho = collections.OrderedDict([('B', []), ('C', []), ('D',
[])])

    # --- FASE 1: DISTRIBUIÇÃO ---
    while A:
        numero_atual = A.pop(0)
        colocado = False
        tentativas_movimento = 0

        while not colocado:
            nome_lista_destino, posicao =
encontrar_lugar_para_inserir(numero_atual, listas_trabalho)

            if nome_lista_destino:
                if posicao == 'inicio':
                    listas_trabalho[nome_lista_destino].insert(0, numero_atual)
                else:
                    listas_trabalho[nome_lista_destino].append(numero_atual)
                colocado = True
            else:
                movimento_feito =
mover_entre_listas_para_abrir_espaco(listas_trabalho, set())
                if not movimento_feito:
                    break
                tentativas_movimento += 1
                if tentativas_movimento > 10: # Salvaguarda
                    break

    # --- FASE 2: CONSOLIDAÇÃO ---
    return consolidar_listas(listas_trabalho)

# Exemplo de execução
A_inicial = [3, 5, 6, 2, 4, 7, 8, 9, 13, 1, 12, 11, 10]
lista_final = ordenar_cartas(list(A_inicial))
print(f"Lista Final Ordenada: {lista_final}")
print(f"Verificação (sorted): {sorted(A_inicial)}")
```

REFERÊNCIAS

1. Time and Space Complexity analysis of Stack operations - GeeksforGeeks, accessed August 16, 2025, https://www.geeksforgeeks.org/dsa/time-and-space-complexity-analysis-of-stack-operations/
2. Deque in Algorithm Design: A Deep Dive - Number Analytics, accessed August 16, 2025, https://www.numberanalytics.com/blog/deque-in-algorithm-design-deep-dive
3. Mastering Deque in Algorithm Design - Number Analytics, accessed August 16, 2025, https://www.numberanalytics.com/blog/ultimate-guide-to-deque-in-algorithm-design
4. Deque Data Structure - Programiz, accessed August 16, 2025, https://www.programiz.com/dsa/deque
5. artigo-ordenacao-manual.docx
6. Sort a Stack Using Temporary Stack - EnjoyAlgorithms, accessed August 16, 2025, https://www.enjoyalgorithms.com/blog/sort-stack-using-temporary-stack/
7. time complexity for a sorting technique for stacks? - Stack Overflow, accessed August 16, 2025, https://stackoverflow.com/questions/34384823/time-complexity-for-a-sorting-technique-for-stacks
8. Time Complexity of Insertion Sort - Stack Overflow, accessed August 16, 2025, https://stackoverflow.com/questions/19827193/time-complexity-of-insertion-sort
9. Analysis of selection sort (article) | Khan Academy, accessed August 16, 2025, https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort
10. Analysis of Insertion Sort - Computer Science, accessed August 16, 2025, http://www.cs.albany.edu/~petko/classes/FALL16-CSI403-slides/02-Analysis%20of%20Insertion%20Sort.pdf
11. Insertion sort - Wikipedia, accessed August 16, 2025, https://en.wikipedia.org/wiki/Insertion_sort
12. Insertion Sort Explained–A Data Scientists Algorithm Guide | NVIDIA Technical Blog, accessed August 16, 2025, https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/
13. Time and Space Complexity of Insertion Sort - GeeksforGeeks, accessed August 16, 2025, https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-insertion-sort-algorithm/
14. Understanding Insertion Sort: A Step-by-Step Guide with Complexity Analysis - Medium, accessed August 16, 2025, https://medium.com/coffee-and-codes/understanding-insertion-sort-a-step-by-step-guide-with-complexity-analysis-54592340a3c1
15. Selection Sort Algorithm: Time and Space Complexity Analysis - Youcademy, accessed August 16, 2025, https://youcademy.org/selection-sort-complexity/
16. All Types of Sorting Algorithms in Data Structure (With Examples) - WsCube Tech,

accessed August 16, 2025, https://www.wscubetech.com/resources/dsa/sorting-algorithms
17. Selection sort - Wikipedia, accessed August 16, 2025, https://en.wikipedia.org/wiki/Selection_sort
18. Exploring the Best Sorting Algorithms for Efficient Data Management - AlgoCademy, accessed August 16, 2025, https://algocademy.com/blog/exploring-the-best-sorting-algorithms-for-efficient-data-management/
19. Selection Sort - GeeksforGeeks, accessed August 16, 2025, https://www.geeksforgeeks.org/dsa/selection-sort-algorithm-2/
20. Quicksort - Wikipedia, accessed August 16, 2025, https://en.wikipedia.org/wiki/Quicksort
21. Quick Sort - GeeksforGeeks, accessed August 16, 2025, https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/
22. Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks, accessed August 16, 2025, https://www.geeksforgeeks.org/dsa/merge-sort/
23. Merge sort - Wikipedia, accessed August 16, 2025, https://en.wikipedia.org/wiki/Merge_sort
24. Sorting algorithm - Wikipedia, accessed August 16, 2025, https://en.wikipedia.org/wiki/Sorting_algorithm