A Régua de Avaliação do Projeto Push Swap (42 SP): Critérios de Conformidade e Otimização Algorítmica

I. Sumário Executivo: A Natureza Dupla da Avaliação

O projeto push_swap na Escola 42 é um desafio algorítmico complexo focado na otimização da ordenação de dados. O objetivo principal do projeto é desenvolver um programa que possa ordenar uma pilha de inteiros (denominada Pilha A) em ordem ascendente, utilizando exclusivamente um conjunto restrito de 11 operações e uma pilha auxiliar (Pilha B), com a condição primária de que essa ordenação seja alcançada com o **menor número possível de instruções** .

A régua de avaliação utilizada pela 42 Network é rigorosamente estruturada em dois pilares distintos. O primeiro pilar é o da **Conformidade Mandatória**, que funciona como um "Zero Gate": o projeto deve demonstrar robustez técnica, gestão de memória perfeita e tratamento de erros rigoroso. A falha em qualquer um destes critérios eliminatórios resulta em uma nota de 0 (zero), independentemente da eficácia do algoritmo de ordenação implementado.[1]

O segundo pilar, e o mais desafiador, é a **Otimização Algorítmica**. Este pilar quantifica a eficiência do programa pela contagem de movimentos (instruções) necessários para ordenar a pilha. A pontuação final é diretamente correlacionada à capacidade do aluno de se aproximar dos limites de contagem de movimentos que caracterizam soluções de alta complexidade e eficiência (tipicamente $O(N \log N)$), especialmente para grandes volumes de entrada ($N=100$ e $N=500$) . A estratégia de desenvolvimento, portanto, exige que a robustez e a higiene de código sejam priorizadas, seguidas pela seleção e otimização de um algoritmo de ordenação escalável.

II. Conformidade Mandatória: O Requisito de "Zero" (O Gatekeeping da Régua)

A etapa inicial e não negociável da avaliação de push_swap é a verificação da conformidade com os requisitos de robustez e segurança. Esses critérios são eliminatórios e refletem a proficiência técnica fundamental do estudante no desenvolvimento em C.

2.1. Tratamento de Erros (Error Handling)

O programa push_swap deve ser excepcionalmente robusto na validação dos argumentos de entrada.[3] Qualquer falha na validação deve ser tratada através da emissão estrita da mensagem "Error\n" para o *standard output* e o encerramento imediato do programa.[3]
A validação de entrada abrange três áreas críticas:

1. **Validação de Formato e Caracteres:** O programa deve aceitar números inteiros como argumentos separados por espaço na linha de comando ou, em algumas variantes, como uma única *string* contendo números.[3] A inserção de quaisquer caracteres não numéricos ou formatos ambíguos (ex: ./push_swap 1 3 dog 35 80 -3) deve ser rejeitada imediatamente.[3]
2. **Verificação de Limites de Tipo (INT Limits):** Como o projeto lida com inteiros, o programa deve ser capaz de detectar *overflow* e *underflow* em relação aos limites de um inteiro de 32 bits (tipicamente $2,147,483,647$ a $-2,147,483,648$). Entradas que excedam esses limites (ex: 54867543867438) devem gerar um erro.[3] É crucial que o código utilize tipos de dados mais amplos, como long int ou long long int, durante a fase de *parsing* e validação, para verificar se o valor lido se encaixa nos limites do int antes de ser aceito.[3]
3. **Validação de Duplicatas:** Um requisito essencial da régua é que a entrada não contenha números duplicados. O programa deve verificar e rejeitar qualquer conjunto de argumentos que inclua repetições, como no exemplo ./push_swap 1 3 58 9 3 .

2.2. Gestão de Memória (Memory Leaks)

A ausência de vazamentos de memória é um requisito mandatório absoluto. O uso da ferramenta Valgrind (ou *testers* automatizados similares [4]) é padrão na 42 para verificar a higiene de código.[1] O projeto, que geralmente exige a construção de estruturas de dados dinâmicas (frequentemente listas duplamente encadeadas circulares para simular pilhas e otimizar as rotações), deve ser executado em testes extensivos sem reportar qualquer vazamento de memória.[1]
A exigência de Error Handling e ausência de leaks zero impõe uma dificuldade estrutural no projeto. Essa necessidade força o aluno a escrever código modular e defensivo desde o início. Se a fundação (como a implementação da lista encadeada e a lógica de *parsing*) apresentar falhas de segmentação ou vazamentos, qualquer esforço subsequente na otimização do algoritmo de ordenação será inútil, resultando na nota zero mandatória.[1] A complexidade do projeto reside não apenas no algoritmo de ordenação, mas na construção de estruturas de dados robustas que gerenciem a memória perfeitamente e validem entradas de maneira exaustiva.[3]

2.3. O Programa Checker (Parte Bônus/Validação)

A avaliação do projeto é complementada pelo desenvolvimento de um programa checker . Este programa recebe a lista inicial de números, lê a lista de instruções geradas pelo push_swap e executa essas operações nas pilhas.[7] Sua função é verificar a validade do resultado final: a Pilha A deve estar perfeitamente ordenada em ordem crescente, e a Pilha B deve estar vazia.[8] O checker reporta "OK" ou "KO".[7] O próprio checker deve ser robusto, tratando erros de argumentos malformados ou instruções inválidas .
Tabela I: Requisitos de Conformidade Mandatória (Checklist)

Critério Mandatório	**Requisito Mínimo (Zero Gate)**	**Implicação**	**Referência**
Erro: Não Numérico/Inválido	Rejeitar entradas não estritamente numéricas.	Requisito de *parsing* rigoroso.	[3]
Erro: Limites de INT	Rejeitar números que causem *overflow/underflow* de 32 bits.	Necessidade de manipulação de long int para validação.	[3]
Erro: Duplicatas	Rejeitar qualquer conjunto de números com repetição.	Implementação de lógica de verificação de unicidade.	
Gestão de Memória	Zero Memory Leaks (verificado por Valgrind).	Uso estrito de free() e estruturas de dados apropriadas.	[1, 4, 2, 6]
Pilha Final	Pilha A ordenada em ordem ascendente; Pilha B vazia.	Requisito de estado final do algoritmo.	[8]

III. O Conjunto de Instruções e a Economia de Movimentos

A régua de otimização é fundamentada na contagem das instruções atômicas geradas pelo programa push_swap. O aluno deve dominar o uso eficiente das 11 operações permitidas.

3.1. Análise Funcional das Operações Permitidas

O conjunto de instruções é o vocabulário limitado que o algoritmo deve usar para ordenar a

Pilha A :

1. **Swap (s):** Troca os dois primeiros elementos no topo da pilha.
    - sa (Pilha A), sb (Pilha B), ss (Simultâneo em A e B) .
2. **Push (p):** Transfere o elemento do topo de uma pilha para o topo da outra.
    - pa (B para A), pb (A para B) .
3. **Rotate (r):** Move todos os elementos para cima por uma posição, o primeiro elemento se torna o último.
    - ra (Pilha A), rb (Pilha B), rr (Simultâneo em A e B) .
4. **Reverse Rotate (rr):** Move todos os elementos para baixo por uma posição, o último elemento se torna o primeiro.
    - rra (Pilha A), rrb (Pilha B), rrr (Simultâneo em A e B) .

3.2. A Importância Estratégica das Operações Simultâneas

A métrica de avaliação valoriza a **minimização da contagem total de movimentos** . Neste contexto, a utilização estratégica das operações simultâneas (ss, rr, rrr) é determinante. Cada uma dessas operações conta como uma única instrução, embora execute duas ações de forma coordenada.[8]

Em fases críticas do algoritmo, especialmente durante a otimização de movimentos de retorno (da Pilha B para A), é frequentemente necessário posicionar tanto o elemento alvo no topo de B quanto o local de inserção correto no topo de A. Se ambos os movimentos (em A e B) puderem ser realizados na mesma direção (ambos rotate ou ambos reverse rotate), a execução de rr ou rrr economiza um movimento em comparação com a execução sequencial de ra seguido de rb, por exemplo. Um algoritmo de pontuação máxima deve, portanto, integrar uma Análise de Custo (Cost Analysis) que priorize explicitamente o uso coordenado de rr ou rrr para atingir os limites mais baixos da régua.

Tabela II: Detalhamento das Instruções Operacionais

Instrução	**Descrição Funcional**	**Contagem**	**Efeito na Pilha**
sa, sb, ss	Troca dos 2 elementos do topo.	1 movimento	Local ou Conjunto (topo)
pa, pb	Transfere o elemento do topo de uma pilha para a outra.	1 movimento	Inter-pilhas
ra, rb, rr	Rotação: topo vai para a base (Shift Up).	1 movimento	Estrutural (shift up)
rra, rrb, rrr	Rotação Reversa: base vai para o topo (Shift Down).	1 movimento	Estrutural (shift down)

IV. A Régua de Otimização: Tiers de Pontuação Baseados em Movimentos

A métrica central da régua de avaliação é a eficiência quantitativa. O projeto é testado com tamanhos de pilha crescentes (N), e o número de movimentos gerados pelo programa é comparado a *tiers* de pontuação predefinidos.[10]

4.1. Requisitos para Pilhas Pequenas ($N=3$ e $N=5$)

Os casos de $N=3$ e $N=5$ são cruciais, pois testam a capacidade do aluno de implementar lógica de ordenação especializada com complexidade $O(1)$ — ou seja, com um número fixo e mínimo de movimentos.

- **$N=3$:** O algoritmo é obrigado a ordenar a pilha em **3 operações ou menos** . Soluções ideais atingem 1 a 2 movimentos , exigindo a identificação e tratamento explícito dos 6 casos de permutação possíveis .
- **$N=5$:** O requisito é ordenar a pilha em **12 operações ou menos** . Soluções eficientes devem atingir menos de 10 movimentos, com relatórios indicando que o ideal é atingir ≤ 8 movimentos . A estratégia comum envolve empurrar os dois menores números para B, ordenar os três restantes em A, e depois retornar os números de B para A de forma otimizada .

4.2. Tiers de Pontuação para Pilhas Grandes ($N=100$ e $N=500$)

Para pilhas maiores, a régua avalia a escalabilidade do algoritmo, impondo limites que só podem ser alcançados por soluções de complexidade logarítmica ou próxima a ela.

N = 100

Tamanho (N)	Max Score (Target)	Threshold (Alta Pontuação)	Pontuação Mínima (Falha)	Referência
N=100	≤ 700 movimentos	< 1300 movimentos	> 1500 movimentos	

Para $N=100$, o limite de 1300 movimentos é frequentemente citado como o ponto de corte para uma pontuação alta (aproximadamente 4/5 pontos) . O objetivo de pontuação máxima é ≤ 700 movimentos . O desempenho médio esperado para um algoritmo $O(N

\log N)$ bem implementado fica tipicamente na faixa de 855 a 857 operações.[5]

N = 500

Tamanho (N)	Max Score (Target)	Threshold (Alta Pontuação)	Pontuação Mínima (Falha)	Referência
N=500	≤ 5500 movimentos	< 10000 movimentos	> 11500 movimentos	

O teste com $N=500$ é a prova de fogo da eficiência. O limite superior de 11500 movimentos penaliza fortemente algoritmos ineficientes . A pontuação máxima é reservada para soluções que atingem ≤ 5500 movimentos . Resultados da comunidade demonstram que algoritmos otimizados alcançam consistentemente médias próximas de 5739-5741 movimentos [5], e em alguns casos, resultados abaixo de 4200 .

4.3. A Imposição da Complexidade Algorítmica

A estrutura da régua de movimentos impõe implicitamente um requisito de complexidade algorítmica. Não se trata apenas de ordenar, mas de usar a menor quantidade de recursos (instruções). Um algoritmo de ordenação quadrática ($O(N^2)$), como Bubble Sort ou Insertion Sort, falharia drasticamente, pois sua contagem de movimentos para $N=500$ seria proibitivamente alta, ultrapassando facilmente o limite de 11500.[10] Para satisfazer os *tiers* de pontuação máxima, o aluno deve implementar uma solução com complexidade próxima a $O(N \log N)$.[6] A diferença entre um desempenho $O(N \log N)$ e $O(N^2)$ para $N=500$ é a diferença entre a aprovação e a reprovação no aspecto de otimização da régua.

V. Estratégias Algorítmicas para Otimização (O Caminho para o Max Score)

Para atingir os rigorosos limites de movimentos, o aluno deve adotar e otimizar um algoritmo que utilize eficientemente o espaço das duas pilhas, transcendendo as técnicas tradicionais de ordenação por comparação.

5.1. Normalização de Dados (Ranking)

O primeiro passo, fundamental para a aplicação de algoritmos eficientes baseados em bit ou

índice, é a normalização dos dados. Os valores inteiros brutos de entrada (que podem variar de INT_MIN a INT_MAX) devem ser convertidos em seus respectivos *ranks* ou índices, variando de 0 a $N-1$. Esta conversão permite que o algoritmo de ordenação trabalhe com a posição relativa dos números, ignorando seus valores absolutos, o que é essencial para o sucesso do Radix Sort adaptado.[4]

5.2. A Estratégia Predominante: Radix Sort Adaptado (Base 2)

O Radix Sort é amplamente reconhecido como a abordagem mais eficiente para cumprir as exigências de $O(N \log N)$ da régua.[6] O algoritmo é adaptado para operar em base 2, utilizando as duas pilhas A e B como os dois "baldes" binários necessários para a ordenação.[5] O processo funciona iterando sobre cada bit dos *ranks* normalizados, começando pelo bit menos significativo [5]:

1. **Iteração:** Para o bit atual, o algoritmo percorre todos os elementos da Pilha A.
2. **Push (0):** Se o bit atual do número for 0, o número é transferido para a Pilha B utilizando a instrução pb.[5]
3. **Rotate (1):** Se o bit atual for 1, o número permanece na Pilha A, e a pilha é rotacionada (ra) para expor o próximo elemento.[5]
4. **Consolidação:** Após varrer todos os N elementos, a Pilha A contém apenas os números que tinham o bit 1 e a Pilha B contém os números com o bit 0. Todos os elementos da Pilha B são então movidos de volta para a Pilha A utilizando pa antes de passar para o próximo bit mais significativo.

A complexidade temporal resultante, $O(k \cdot N)$, onde k é o número de bits necessários para representar N números (aproximadamente $\log_2 N$), satisfaz perfeitamente os requisitos de performance para $N=100$ e $N=500$, garantindo que a contagem de movimentos permaneça dentro dos limites de pontuação máxima.[6]

5.3. Análise de Custo (Cost Analysis) para o Retorno Otimizado

A otimização mais crítica e que distingue um bom algoritmo Radix Sort de um algoritmo de pontuação máxima ocorre durante a fase de retorno dos números da Pilha B para a Pilha A (o push back) .
Neste estágio, o algoritmo deve determinar o número mínimo de movimentos necessários para transferir o elemento ideal do topo de B para sua posição correta na Pilha A, garantindo que A permaneça ordenada ou quase ordenada. Isso exige um cálculo de custo de rotação que compare as quatro combinações possíveis de movimentos de alinhamento:

1. **Rotação Conjunta (rr):** Usar rr para girar A e B simultaneamente na direção normal (para cima), seguido de rotações individuais.
2. **Rotação Reversa Conjunta (rrr):** Usar rrr para girar A e B simultaneamente na direção reversa (para baixo), seguido de rotações individuais.

3. **Combinação Mista I:** ra e rrb (rotações em direções opostas).
4. **Combinação Mista II:** rra e rb (rotações em direções opostas).

A diferença entre um algoritmo que simplesmente usa rotações individuais sem coordenação e um que calcula o caminho mais curto, priorizando rr/rrr quando possível, é o fator decisivo para atingir a zona de Max Score (≤ 700 para $N=100$).[10] A integração de uma análise de custo para determinar o caminho mais eficiente é a otimização final exigida pela régua para evitar o excesso de movimentos .

VI. O Protocolo de Avaliação e o Checker

O processo de avaliação é formalizado pelo protocolo de *peer review* da 42, com a validação final sendo conduzida pelo programa checker.[1]

6.1. Validação Automática da Saída

O push_swap produz uma sequência de instruções no *standard output*. Esta sequência é então canalizada para o checker . Se o checker executar todas as instruções e o estado final for Pilha A ordenada e Pilha B vazia, a saída é "OK".[7]
O número de instruções geradas é então contabilizado. É esta contagem que é comparada com os *tiers* de pontuação da Tabela III. A pontuação não é baseada em um único caso de teste, mas sim na performance média do algoritmo em múltiplos *inputs* aleatórios de um determinado tamanho.[5] A avaliação, portanto, exige que o algoritmo mantenha um desempenho $O(N \log N)$ de forma consistente, mesmo em cenários de *stress* de dados.[2]
Tabela III: Régua Quantitativa de Otimização (Tiers de Movimentos)

Tamanho (N)	Complexidade Requerida	Max Score (Target)	Threshold 4/5 Pontos	Pontuação Mínima (KO)	Referência
N=3	$O(1)$	≤ 3	N/A	> 3	
N=5	$O(1)$	≤ 12 (Ideal ≤ 8)	N/A	> 12	
N=100	$O(N \log N)$	≤ 700	< 1300	> 1500	
N=500	$O(N \log N)$	≤ 5500	< 10000	> 11500	

VII. Conclusão e Recomendações Estratégicas

A régua de avaliação do projeto push_swap na 42 SP é uma ferramenta de medição sofisticada que avalia tanto a disciplina de engenharia de software quanto a competência

algorítmica. O sucesso é definido pela superação sequencial dos dois pilares de avaliação.
1. Garantia da Conformidade Mandatória (O "Zero Gate"): A recomendação estratégica primária é investir tempo substancial na implementação de um *parsing* de argumentos impecável, com tratamento completo de erros para duplicatas, limites de inteiros (32 bits) e formatos inválidos.[3] Concomitantemente, a gestão de memória deve ser rigorosa para garantir *leaks* zero, evitando que falhas básicas de higiene de código anulem o trabalho de otimização.[1]
2. Implementação de Algoritmo Escalável ($O(N \log N)$): Para pilhas grandes ($N=100$ e $N=500$), a única via para satisfazer os limites de movimentos é a adoção de um algoritmo de baixa complexidade, como o Radix Sort adaptado a Base 2.[5] A conversão inicial dos inteiros para *ranks* é um pré-requisito técnico para esta abordagem.[10]
3. Otimização Final através de Análise de Custo: Para alcançar os *tiers* de pontuação máxima (Max Score), o algoritmo deve integrar uma análise de custo na fase de retorno dos elementos (B $\rightarrow$ A) . Esta análise deve calcular e executar o caminho mais curto possível, maximizando o uso das operações simultâneas (rr e rrr) para reduzir a contagem final de instruções abaixo dos limiares de 700 e 5500 movimentos . A capacidade de otimizar esta fase é o fator que diferencia as soluções aceitáveis das soluções de excelência na régua da 42 Network.

Referências citadas

1. Push Swap | PDF | Computer Engineering - Scribd, acessado em novembro 2, 2025, https://www.scribd.com/document/667387295/push-swap-1
2. GitHub topics: push-swap-tester | Ecosyste.ms - Repos, acessado em novembro 2, 2025, https://repos.ecosyste.ms/hosts/GitHub/topics/push-swap-tester
3. Building the thing | Guide, acessado em novembro 2, 2025, https://42-cursus.gitbook.io/guide/2-rank-02/push_swap/building-the-thing
4. hu8813/tester_push_swap: A simple Push_swap tester for testing memory leaks/errors and error handling for the 42 school project pushswap - GitHub, acessado em novembro 2, 2025, https://github.com/hu8813/tester_push_swap
5. hu8813/push_swap: 42 school project pushswap using radix sort - GitHub, acessado em novembro 2, 2025, https://github.com/hu8813/push_swap
6. Lowest possible estimated number of steps or time complexity for push_swap program for 42 - Stack Overflow, acessado em novembro 2, 2025, https://stackoverflow.com/questions/77228485/lowest-possible-estimated-number-of-steps-or-time-complexity-for-push-swap-progr
7. liz753/push_swap: 42 Project about sorting algorithms. The program, called push_swap, takes a list of integers and sorts it using a set of predefined operations on two stacks. Another program, called checker, verifies the correctness of the sorting by executing the instructions generated by push_swap on the stack A. - GitHub, acessado em novembro 2, 2025, https://github.com/liz753/push_swap
8. push_swap | Guide - GitBook, acessado em novembro 2, 2025, https://42-cursus.gitbook.io/guide/2-rank-02/push_swap

9. Push Swap in less than 4200 operations | by Ulysse - Medium, acessado em novembro 2, 2025, https://medium.com/@ulysse.gks/push-swap-in-less-than-4200-operations-c292f034f6c0
10. Push_swap: what is the most efficient way to sort given values using a limited set of instructions on two rotatable stacks?, acessado em novembro 2, 2025, https://stackoverflow.com/questions/75100698/push-swap-what-is-the-most-efficient-way-to-sort-given-values-using-a-limited-s