Uma análise da coluna Cracking the Oyster

Giovanni Milan Câmara Pinto, João Henrique Alves Silva, e Samuel Alves Gomes Palaoro

Universidade Vila Velha

2026-02-13

Resumo

Este artigo tem como objetivo analisar a coluna *Cracking the Oyster* do livro *Programming Pearls*, de Jon Bentley. O texto apresenta o desafio de ordenar até dez milhões de inteiros distintos sobre a restrição de usar apenas 1MB de memória, a ordenação deve ser rápida e eficiente.Serão discutidas as soluções inicialmente consideradas e a alternativa final utilizada pelos programadores, que se mostrou eficiente e adequada às limitações da época.

Sumário

1 Introdução

A coluna *Cracking the Oyster*, do livro *Programming Pearls*, de Jon Bentley, descreve um estudo de caso sobre a importância de formular corretamente um problema antes de buscar sua solução. O autor apresenta a situação de um programador que precisa ordenar até dez milhões de números inteiros utilizando apenas um megabyte de memória disponível. A partir disso, ele discute abordagens tradicionais de ordenação e explica por que elas não são viáveis para esse caso específico, devido às limitações da época, conduzindo o leitor à solução definitiva. O estudo mostra que, além de revelar uma estratégia eficiente para o problema, também oferece lições valiosas sobre design de algoritmos e economia de recursos.

2 Revisão bibliográfica

Os problemas apresentados na coluna *Cracking the Oyster* envolvem conceitos fundamentais de algoritmos de ordenação e de estruturas de dados eficientes sob restrições de memória. Algoritmos como *Merge Sort* e *QuickSort* são amplamente utilizados em ordenação interna, isto é, quando todos os elementos podem ser carregados simultaneamente na memória principal. No entanto, quando o conjunto de dados ultrapassa o limite de memória disponível, torna-se necessário recorrer a técnicas de ordenação externa, nas quais parte do processamento depende do disco. Embora o *Merge Sort* seja bastante utilizado nesse contexto devido à sua estabilidade e bom desempenho, ele não é adequado para o caso analisado, pois sua aplicação envolve a criação de arquivos intermediários e múltiplas leituras, resultando em um tempo de execução elevado em sistemas com hardware limitado.

Outra abordagem discutida no livro é a técnica *multipass*, na qual os dados são divididos em faixas numéricas e várias leituras sequenciais são realizadas para filtrar e ordenar cada intervalo. Contudo, embora essa técnica reduza a necessidade de memória principal, ela aumenta significativamente o número de acessos ao disco, tornando o processo lento quando o arquivo é muito grande ou quando há exigência de alto desempenho.

A coluna também explora o uso de estruturas de dados compactas, como os vetores de bits. Essa técnica é amplamente utilizada em estudos sobre estruturas eficientes para conjuntos densos, nos quais cada elemento pertence a um intervalo numérico bem definido. O *bitmap* permite representar a presença ou ausência de cada valor utilizando apenas um bit por elemento possível, sendo, portanto, uma alternativa extremamente econômica em memória e adequada para problemas de ordenação sem repetição de valores.

Esses conceitos ordenação externa, algoritmos *multipass* e estruturas compactas como *bitmaps* constituem a base teórica necessária para compreender as soluções avaliadas na coluna e justificam por que a abordagem final se mostrou mais eficiente do que as demais no contexto analisado.

3 Metodologia

3.1 Definição do problema

A metodologia apresentada na coluna inicia-se pela formulação correta do problema enfrentado pelo programador. O desafio consiste em ordenar até dez milhões de números inteiros distintos, todos pertencentes a um intervalo de sete dígitos, sob a limitação computacional de utilizar aproximadamente um megabyte de memória principal. Além disso, a ordenação precisava ser executada em um tempo reduzido, o que inviabilizava soluções que dependessem de múltiplas leituras do disco.

3.2 Análise das soluções tradicionais

A primeira etapa do processo consiste em avaliar soluções tradicionais de ordenação. O *Merge Sort* em disco foi inicialmente considerado; contudo, sua aplicação resultaria na criação de diversos arquivos

intermediários e em sucessivas operações de leitura e escrita, elevando consideravelmente o tempo de execução. Em seguida, foi analisada a técnica de *Multipass*, que divide os dados em faixas numéricas e realiza diversas passagens sobre o arquivo. Entretanto, esse método poderia exigir cerca de quarenta leituras completas, o que é extremamente custoso em termos de operações de *I/O*. Além disso, se o intervalo numérico fosse amplo demais, a memória não suportaria sua representação; se fosse estreito, o número de passagens aumentaria significativamente. Métodos de ordenação interna, como o *QuickSort*, também foram descartados, pois exigem carregar todos os números simultaneamente na memória, ultrapassando o limite disponível.

3.3 Identificação da estrutura adequada

A etapa seguinte consistiu em observar as características do conjunto de dados. Como todos os inteiros pertenciam ao intervalo de 0 a 9.999.999 e não havia duplicatas, tornou-se desnecessário armazenar os números em si. Dessa forma, bastava representar a presença ou ausência de cada valor possível, o que motivou a adoção de uma estrutura de dados compacta: o vetor de bits.

3.4 Implementação da solução com bitmap

A solução final consiste na criação de um vetor de dez milhões de bits, inicialmente zerados. Em uma única leitura do arquivo de entrada, para cada número x encontrado, o bit correspondente é marcado como 1. Após o processamento de todos os valores, o vetor é percorrido sequencialmente, e o índice de cada posição cujo bit esteja ativado é escrito no arquivo de saída. Como a varredura ocorre em ordem crescente, a saída já se encontra totalmente ordenada, dispensando qualquer comparação entre elementos.

Listing 1: Pseudocódigo do algoritmo de ordenação com bitmap

```
/* Fase 1: inicializar conjunto vazio */
for i = 0 to n-1
    bit[i] = 0

/* Fase 2: marcar valores presentes no arquivo de entrada */
for each x in input_file
    bit[x] = 1

/* Fase 3: escrever saída ordenada */
for i = 0 to n-1
    if bit[i] == 1
        write i to output_file
```

3.5 Justificativa da eficiência

O uso de um vetor de bits permite representar o conjunto de valores utilizando a quantidade mínima de memória, alocando apenas um bit por número potencial. Assim, a ordenação reduz-se a um processo linear de marcação e varredura, eliminando a necessidade de arquivos intermediários, comparações e múltiplas passagens sobre o disco. Essa abordagem mostrou-se altamente eficiente mesmo diante das restrições impostas, consumindo aproximadamente *1,25MB* de memória e apresentando tempo de execução reduzido.

4 Conclusão

A análise da coluna *Cracking the Oyster* evidenciou que a formulação correta de um problema é tão importante quanto a escolha do algoritmo utilizado para solucioná-lo. O caso apresentado demonstra

que, mesmo em situações nas quais abordagens tradicionais de ordenação são amplamente conhecidas e eficientes, elas se tornam inviáveis quando aplicadas sob restrições severas de memória e de operações de entrada e saída.

A partir do entendimento adequado do problema, foi possível adotar uma solução simples e altamente eficaz para a ordenação de um conjunto de números inteiros sem repetição: o uso de um vetor de bits. Essa abordagem eliminou a necessidade de comparações entre elementos, da criação de arquivos intermediários e de múltiplas leituras de disco, reduzindo todo o processo de ordenação a uma operação linear, compatível com a limitação de memória disponível.

Assim, a coluna reforça lições fundamentais sobre eficiência, economia de recursos e a importância do bom design de algoritmos, demonstrando que soluções elegantes frequentemente surgem da compreensão profunda das características e restrições do problema.