Pérolas da Programação: As Ordenações

Uma análise sobre a Coluna 11 de Jon Bentley

Andersson Santos Ferreira
Caio Alves Martins de Souzas
Davi André de Carvalho Pereira

Universidade Vila Velha (UVV)

06 de Março de 2026

Resumo

Este trabalho apresenta um resumo analítico da Coluna 11 da obra clássica Programming Pearls, de Jon Bentley, dedicada ao estudo de algoritmos de ordenação e suas otimizações práticas. O foco central é demonstrar como a escolha adequada de algoritmos e pequenas melhorias na implementação podem gerar ganhos significativos de desempenho. A partir da análise de técnicas como Insertion Sort e Quicksort, o autor discute estratégias como particionamento eficiente, melhor escolha do pivô, uso de algoritmos híbridos e redução de custos de recursão. O capítulo também enfatiza a importância de medir o desempenho real dos programas e de utilizar implementações de bibliotecas quando possível. Conclui-se que, além da teoria dos algoritmos, a atenção aos detalhes de implementação e às otimizações práticas é fundamental para o desenvolvimento de softwares eficientes.

Sumário

1 Introdução

A ordenação de dados é uma das tarefas mais comuns em computação. Diversos sistemas dependem da organização eficiente de informações. Para esse propósito vários métodos foram desenvolvidos, sendo aplicados em bancos de dados, sistemas de busca e processamento de grandes volumes de informação. Na Coluna 11 de Programming Pearls, Jon Bentley explora o conceito da ordenação sob uma perspectiva prática. Reconhecendo que embora muitas linguagens possuem funções prontas para ordenar dados, existem situações em que o programador precisa implementar seu próprio método para obter melhor desempenho ou atender requisitos específicos.

2 Insertion Sort

O Insertion Sort é um algoritmo simples inspirado na forma como jogadores de cartas organizam suas mãos. Cada novo elemento é inserido na posição correta dentro do array mantendo este ordenado.

2.1 Implementação

Dado um vetor X contendo n elementos, deseja-se reorganizá-los de modo que fiquem em ordem crescente.

```
X[0]   X[1]   X[2]   ...   X[n-1]
for i = 1 até n-1
Inserir X[i] na posição correta entre X[0..i-1]


for (i = 1; i < n; i++) {
        t = x[i];
        for (j = i; j > 0 && x[j-1] > t; j--) {
            x[j] = x[j-1];}
        x[j] = t;}
```

2.2 Vantagens

O resultado é um método de ordem $O(n^2)$ no pior caso, apresentando vantagens de simplicidade de implementação, baixo overhead e bom desempenho em listas pequenas ou quase ordenadas.

3 Quicksort

O Quicksort é um dos algoritmos de ordenação mais eficientes na prática. Ele utiliza a estratégia de divisão e conquista, particionando o array em duas partes em torno de um elemento chamado pivô.

3.1 Particionamento do Array

Durante o particionamento o array é reorganizado em torno do pivô. Todos os elementos menores que o pivô ficarão à sua esquerda, e todos os elementos maiores que o pivô ficarão à sua direita.

```
m = a - 1;
for (int i = a; i <= b; i++) {
    if (x[i] < t) {
        m++;
        int temp = x[m];
        x[m] = x[i];
        x[i] = temp;}}
```

3.2 Implementação

O processo é aplicado recursivamente às sub matrizes particionadas resultantes até o surgimento de sub matrizes de um elemento, finalizando a ordenação.

Uso do primeiro elemento com pivô:

```
void qsort1(x[], l, u) {
    if (l >= u) {
        return;
    }
    m = l;
    for (i = l + 1; i <= u; i++) {
        if (x[i] < x[l]) {
            m++;
            int temp = x[m];
            x[m] = x[i];
            x[i] = temp;
        }
    }
    int temp = x[l];
    x[l] = x[m];
    x[m] = temp;
    qsort1(x, l, m - 1);
```

3.3 Casos extremos e Optimização

Devido a tendência do Quicksort de juntar elementos iguais em uma implementação onde o array possui muitos ou todos os valores idênticos, ou em outro caso onde o array já está ordenado, Quicksort pode apresentar desempenho próximo de $O(n^2)$. Isso acontece devido às várias operações desnecessárias feitas nesse caso no processo.

Esta falha pode ser resolvida na alteração do código, fazendo a partição ser escolhida de forma cuidadosa, focando os maiores e menores valores. Esta alteração também traz o benefício de menos trocas aumentando a eficiência do código.

```
randint(l, u) {
    return l + (rand() % (u - l + 1));
}
void swap(x[], a, b) {
    temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
void qsort4(x[], l, u, cutoff) {
    if (u - l < cutoff) {
        return;
    }
    swap(x, l, randint(l, u));
    t = x[l];
```

```
    i = l;
    j = u + 1;
    while (1) {
        do {
            i++;
        } while (i <= u && x[i] < t);
        do {
            j--;
        } while (x[j] > t);
        if (i > j) {
            break;
        }
        temp = x[i];
        x[i] = x[j];
        x[j] = temp;
    }
    swap(x, l, j);
    qsort4(x, l, j - 1, cutoff);
    qsort4(x, j + 1, u, cutoff);
}
```

4 Conclusão

A Coluna 11 de Programming Pearls mostra que enquanto em amostras pequenas é possível simplesmente utilizar as bibliotecas de sorting específicas da linguagem em uso. Para situações complexas pode se provar necessário a criação de funções feitas à mão que se adequam ao que você precisa.

Também exibe o mérito de combinar Quicksort com Insertion Sort para aproveitar o melhor desempenho em diferentes tamanhos de entrada, balanceando complexidade e velocidade.

Igualmente a coluna deixa nítido como pequenos ajustes na implementação podem transformar a eficiência de maneira significante.