CAPÍTULO 1

ALGORITIMOS DE ORDENAÇÃO INTERNA

1 O PROBLEMA DA ORDENAÇÃO

Entre os algoritmos mais importantes destacam-se os algoritmos de ordenação, em inglês *sort*, isto porque a maior parte das tarefas diárias envolve alguma forma de ordenação. Ordenamos registros em bancos de dados, as tarefas do dia, conhecimento em parágrafos, parágrafos em páginas e páginas em sites. Poderíamos estender esta lista para lembrar que ordenamos desde muito cedo, antes mesmo de aprender a contar. Esta tarefa, a ordenação está emaranhada com o desenvolvimento da humanidade. A própria civilização, como conhecemos, deve seu início à agricultura e a percepção da ordem dos ciclos solares. Ao longo da história os processos de ordenação tiveram impacto na ascensão e queda de impérios e empresas.

O *Google* deve seu sucesso a uma lista ordenada criada a partir de um algoritmo de classificação, o *PageRank*. Foi este algoritmo desenvolvido por ***Larry Page*** e ***Sergey Brin*** que permitiu que o *Google* retornasse uma lista de páginas na ordem que melhor atendesse as demandas de busca de um usuário específico e se tornasse uma empresa quase onipresente. A missão do Google era, talvez ainda seja, indexar todo o conhecimento da humanidade, mas seu sucesso comercial se deve a ordem que este conhecimento é apresentado aos seus usuários. A leitora há de concordar comigo que o uso da palavra clientes seria mais adequado na última sentença. Não a uso apenas para manter o *status quo*[1] da internet. Com tal envolvimento na vida diária, uma parte importante das tarefas computacionais incluem alguma forma de ordenação. Esta adoção faz com que os algoritmos de *sort* estejam também entre os algoritmos mais estudados.

[1] Sempre quis usar *status quo* em uma frase.

Estamos sempre em busca de mais eficiência, para os algoritmos já conhecidos, sem abandonar a esperança de encontrar novos algoritmos para solucionar problemas novos ou antigos.

Uma das primeiras regras de otimização, tarefa diária de todos os envolvidos com eficiência, diz que é melhor começar pelos algoritmos de ordenação. Não bastasse isso como justificativa para o estudo dos algoritmos de ordenação. Precisamos ressaltar que os algoritmos de ordenação são excelentes exemplos do uso das técnicas de criação de algoritmos como, por exemplo: *dividir-e-conquistar*, *estruturas de dados compostas* e uso de aleatoriedade na criação de algoritmos.

Fora do mundo acadêmico é raro utilizarmos um dos algoritmos que vamos estudar para ordenar uma lista de números, palavras ou frases. O comum é que o dado que precisa ser ordenado esteja representado por uma estrutura composta de vários campos, um registro. Em inglês os registros são chamados de *record*. Estes *records* são estruturas de dados compostas de um ou mais campos chaves, a partir dos quais realizaremos a ordenação, e outros tantos campos de informação. Nas linguagens de programação C e C++, os *records* podem ser representados por *structs* e objetos. O uso típico destas estruturas para armazenar informações se estende aos bancos de dados e suas tabelas de registros. Em todos os casos, a ordenação deverá ser realizada em conjuntos contendo itens compostos e resolver um problema específico.

> ***Problema:*** *dado um conjunto de dados qualquer, de que forma podemos ordenar estes dados em uma sequência determinada.*

1.1 Conceito

A escolha do algoritmo para a solução de um determinado problema de ordenação irá depender do problema, dos recursos disponíveis e da característica dos dados. A leitora precisa ter em mente que nem sempre o algoritmo mais rápido será a melhor escolha. Pode ser que o custo envolvido na implementação de um

algoritmo mais rápido e os recursos necessários para garantir que ele realmente seja o mais rápido não compense o benefício adquirido. Esta restrição na escolha dos algoritmos é mais perceptível quando a lista de itens que precisa ser ordenada é pequena ou quando esta lista tem um nível de entropia baixo.

Algoritmos diferentes terão eficiência diferente em dados diferentes. Isso quer dizer que um algoritmo pode ser muito rápido em casos onde os dados estão arranjados em um conjunto com pouca entropia e não ter utilidade nenhuma em casos onde os dados estão na ordem inversa daquela que desejamos.

A palavra entropia, importada da matemática e da física, neste conceito se refere ao percentual de dados que está fora da ordem desejada. Sendo assim, vamos considerar que dados que já se encontrem na ordem desejada terão entropia zero dados totalmente ordenados na ordem invertida, terão entropia 1, ou 100%. A entropia também diz respeito a diferença que existe entre os itens do conjunto que desejamos ordenar, se tivermos pouca diferença entre estes itens, a entropia é baixa, se os itens forem muito diferentes, a entropia é alta.

Do ponto de vista dos algoritmos dividimos o problema em duas classes. Os problemas de ordenação interna e externa. Quando nos referimos a ordenação interna, nos referimos a conjuntos de dados que estão residentes na memória do sistema que está rodando o algoritmo. Nestes problemas não existe a necessidade de recorrer a dispositivos como discos rígidos, ou armazenamento remoto. Todos os outros problemas de ordenamento serão considerados com externos e, serão abordados quando for conveniente.

De uma forma um pouco mais formal, o problema de ordenação pode ser definido da seguinte forma:

> ***Problema:*** *dado um conjunto de dados qualquer, representado por* $R = \{r_1, r_2, r_3, ..\}$ *contendo, no mínimo, um campo chave. De tal forma que, o conjunto de chaves será representado por* $K = \{k_1, k_2, k_3, ...\}$ *. Precisamos ordenar o conjunto* R *de tal forma que seus itens estarão em sequência de tal forma que* $\{k_1 \leq k_2 \leq k_3 \ldots.\}$.

Claramente a descrição matemática do problema de ordenação, anteriormente descrita, está diretamente relacionada com a ordenação crescente do conjunto R utilizando como referência o conjunto de chaves K em **ordem crescente**. Poderíamos ter definido o problema em ordem decrescente, sem prejuízo para a definição do problema de ordenação. Para isso tudo que precisaríamos mudar seria a definição da sequência final de tal forma que esta fosse determinada por $\{k_1 \geq k_2 \geq k_3 \ldots.\}$. Ou seja, a ordem, seja ela crescente, ou decrescente, não afeta a caracterização do problema de ordenação apenas o resultado do processo. Chegaremos à solução do problema de ordenação, crescente ou decrescente, permutando itens dentro do conjunto.

A definição que usamos é suficientemente ampla para permitir a existência de itens no conjunto a ser ordenados que possuam o mesmo valor k_n. Dois ou mais registros podem ter chaves iguais. Aqui, é preciso que a leitora considere que esta possibilidade de repetição de chaves de ordenação não pode ser aplicada a todos os problemas que encontraremos. Bancos de dados, por exemplo, não é raro que algumas tabelas, tenham registros com chaves únicas. No universo dos bancos de dados, a tabela, representa o conjunto de itens a ser ordenado.

O problema de ordenação, como descrito até o momento é relativamente simples se considerarmos apenas chaves numéricas. Chaves alfabéticas requerem um conjunto de regras mais complexos devido a existência de letras maiúsculas, minúsculas, pontuação e nomes compostos. A forma de resolver o problema da ordenação alfabética é a definição de métodos, ou funções, especificamente criados para aplicar as regras de ordenação no momento da permutação de itens.

Caso existam valores duplicados no conjunto de chaves, chamaremos de estáveis aos algoritmos de ordenação que, ao final do processo, tenham mantido a ordem inicial dos itens com chaves duplicadas. Infelizmente poucos algoritmos de ordenação rápidos, são também estáveis.

Os algoritmos de ordenação estáveis, quando aplicados sobre conjuntos com chaves duplicadas, irão colocar os itens duplicados na ordem em que eles

aparecem, um em relação ao outro, antes da ordenação. Se não ficou claro, a Figura 1 apresenta um pequeno exemplo de ordenação estável em um conjunto de cartas ordenadas de forma estável por valor e não por naipe.

Figura 1 - Exemplo de ordenação estável com cartas de baralho.

A leitora pode observar que no conjunto não ordenando a Dama de Copas aparece antes da Dama de Espadas e, consequentemente, acabou antes desta depois do processo de ordenação.

Nossa preocupação será entender os algoritmos de ordenação e vamos utilizar para isso a versão mais simples de cada um dos algoritmos escolhidos para este livro. Todos estes algoritmos serão aplicados sobre um *array* de itens numéricos e, quando for o caso, este modelo será expandido para incluir estruturas de dados mais complexas.

1.2 Análise de eficiência

Analisamos algoritmos para ter uma ideia da sua eficiência que seja independente da plataforma escolhida para a sua execução. Queremos saber se um determinado algoritmo será o mais eficiente possível, independente da máquina que o executará. Um algoritmo ruim em uma máquina excelente, ainda será um

algoritmo ruim, apenas rodará mais rápido do que se estivesse em uma máquina ruim. Um algoritmo eficiente será eficiente, não importa a arquitetura. Observe que eu não falei em tempo de execução, este dependerá da máquina, do conjunto, da entropia dos itens e do algoritmo. Em geral, não temos como mudar nenhum destes itens, exceto o algoritmo. Assim, analisamos os algoritmos, com uma visão matemática do problema, justamente para garantir que, caso nada mais seja alterado, tenhamos o resultado no menor tempo possível.

Vamos analisar os algoritmos para caracterizar sua eficiência utilizando uma variação do artigo de Donald Knuth que, de forma prática, estabeleceu as regras para este tipo de classificação (KNUTH, 1976). Esta teoria da análise dos algoritmos se baseia na definição de três pontos relevantes entre todas as possibilidades de uso de um algoritmo: o pior caso possível, o melhor caso possível e o caso médio. Para definir estes casos definimos três funções: $O(f(n))$; $\Omega(f(n)$ e $\Theta(f(\mathrm{n}))$ para representar o pior, o melhor e o caso médio de uso dos algoritmos. E chamaremos este processo de análise assintótica.

A análise assintótica de um algoritmo representado por uma função $f(n)$ refere-se a taxa de crescimento de $f(n)$ à medida que n cresce. Ou seja, analisamos a eficiência do algoritmo à medida que a cardinalidade do conjunto de itens aumenta. Estamos interessados em conjuntos com muitos elementos, cardinalidade muito alta, na casa dos milhões de itens. Assim, um algoritmo cuja taxa de crescimento seja zero, será melhor que um algoritmo cuja taxa de crescimento seja logarítmica. Pelo menos, esta é a regra típica que usaremos para analisar nossos algoritmos.

Imagine, que temos dois algoritmos, um que pode ser representado por uma função linear, $f_1(n) = d \times n + k$, ou seja, para cada n, o tempo necessário para seu processamento será dado pela cardinalidade do conjunto multiplicada por uma constante d qualquer e este resultado somado a outra constante k, usando as mesmas constantes poderíamos ter um algoritmo dado pela função exponencial $f_1(n) = d \times n^2 + k$. Para que não reste dúvidas da diferença no crescimento destas

duas funções e do seu impacto sobre o tempo. Considere que, nas duas funções, $d = k = 5$ e que n pode assumir os valores $1, 2, 3, 4, 5, 6, 7$, se assim for, o Gráfico 1 mostra o crescimento destas duas funções.

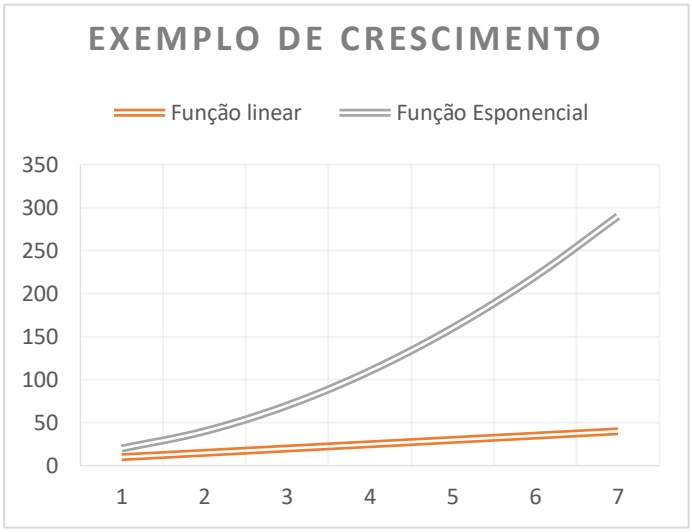

Gráfico 1 - Exemplo de comparação do crescimento de uma função exponencial com uma linear

Se o eixo dos y representar o tempo, com apenas 5 itens, o tempo necessário para execução do algoritmo já é centenas de vezes maior na função exponencial. Eis porque precisamos nos preocupar com a análise de algoritmos.

Funções assintóticas são aquelas que não atingem um determinado valor. Tendem a este valor apenas no infinito. Como não existe o infinito, nunca atingimos este valor. Usamos este nome para este tipo de análise porque vamos analisar o algoritmo no seu pior caso, aquele que irá provocar o maior gasto de tempo. Quando dizermos que um algoritmo roda no tempo $T(n)$ no seu pior caso e, neste caso vamos utilizar a notação $O(f(n))$, chamada, apenas pelos íntimos de notação *Big-O*.

E, usando a notação *Big-O*, ainda usaremos uma técnica de simplificação. Vamos pensar grande, muito grande. Para que a leitora entenda o que é pensar grande, responda, sem pensar muito, quanto custa um automóvel mais um skate? Pode chutar! Não precisa nem procurar na internet. O que queremos é apenas uma aproximação, uma faixa de valores. Que tal R$70.000,00, sem escolher um modelo

específico esta é uma faixa válida? Não? Que tal R$100.000,00? Concorda. Observe que em nenhum momento, nesta consideração, se quer levamos em consideração o preço do skate. Este valor é tão pequeno em relação ao preço de um carro que podemos desprezá-lo completamente e ainda ter uma boa ideia do valor total do investimento. Este é o truque que usaremos na análise assintótica. Vamos nos preocupar apenas com a parte do algoritmo que realmente tem efeito no tempo total de processamento.

Nas funções que usamos como exemplo $f_1(n) = d \times n + k$, e $f_1(n) = d \times n^2 + k$ o impacto das constantes $d\ e\ K$ é tão pequeno que podemos desprezar seu efeito. Em alguns casos, a diferença de tempo será tão grande que não será possível diferenciar o efeito de $d\ e\ K$ no tempo final gasto pelos algoritmos. Assim, recorremos a notação *Big-O,* e estas funções serão respectivamente representadas por: $O(n)$ e $O(n^2)$ que leremos *ordem* n e *ordem* n *ao quadrado*. Sendo assim, o algoritmo representado pela função $f_1(n) = d \times n + k$ é representada por $O(n)$ o que quer dizer que este algoritmo roda em tempo linear. Enquanto, o algoritmo representado por $f_1(n) = d \times n^2 + k$ é representada por $O(n^2)$ o que significa que roda em tempo quadrático. A leitora já deve ter deduzido, mas não custa ressaltar, algoritmos representados por $O(1)$ rodam em tempo constante e independem da cardinalidade do conjunto de itens de entrada.

A principal vantagem da notação *Big-O* é que podemos desconsiderar todos os termos da equação polinomial que represente o algoritmo, por exemplo nas funções $f_1(n) = d \times n + k$, e $f_1(n) = d \times n^2 + k$, podemos desprezar o k antes mesmo de começar. Isto é possível porque a definição original de Knuth explicita que:

$$O\big(g(n)\big) = \{f(n) : constantes\ positivas\ c\ e\ n_0\ tal\ que: 0 \leq f(n) \leq cg(n)\ para\ todos\ n \geq n_0\}$$

A função não negativa $f(n)$ pertence ao conjunto de funções $O(g(n))$ se existir uma constante positiva c que faça $f(n) \leq cg(n)$ para um valor de n suficientemente grande. Sempre é possível escrever $f(n) \in O(g(n))$ porque

$O(g(n))$ é um conjunto. Observe, que na notação assintótica, o sinal de igualdade indica pertencimento e não igualdade. Sendo assim, considere a função $f_3 = 4n^4 + 3n^2 + n + 1237$ usando a notação *Big-O*, podemos desprezar todos os termos cuja ordem seja 4 assim sendo ordem deste algoritmo será dado por $O(n^4)$. Vamos deixar a matemática pesada para um curso de análise de algoritmos. Ainda assim, é necessário destacar os casos mais comuns na notação assintótica e a Tabela 1 assume esta função.

Big-O	Tempo
$O(1)$	Constante
$O(\log n)$	Logarítmico
$O(n)$	Linear
$O(n \log n)$	Linearitimico
$O(n^2)$	Quadrático
$O(n^3)$	Cúbico
$2^{O(N)}$	Exponencial
$O(n!)$	Fatorial

Tabela 1 - Notação Assintótica (Big-O), mais comuns.

Veremos como fazer a análise assintótica dos algoritmos de ordenação a cada um dos algoritmos que analisarmos. Começando pelo mais simples de todos os algoritmos de ordenação.

2 BUBBLE SORT

Bubble Sort, ordenação em bolha, é um dos mais antigos algoritmos de ordenação, em 1962 Kenneth Iverson utiliza este nome em um livro chamado *A Programming Language*[2] e, a partir deste ponto, parece que todos passam a usar este nome para se referir a um processo de ordenação por inversão. Trata-se de

[2] Em tradução livre: uma linguagem de programação. Foi reconfortante encontrar uma versão de *A Programming Language* de Iverson preservada na internet em: *http://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE*

um algoritmo simples cuja única utilidade real é demonstrar o processo de ordenação e explicar como podemos fazer a análise assintótica de um algoritmo. Ainda assim, insistentemente, este algoritmo continua aparecendo nos livros de algoritmos e programação.

Trata-se de um algoritmo de ordenação baseado em comparação. Em cada ciclo, o algoritmo compara dois elementos adjacentes no conjunto que está sendo ordenado e troca a posição daqueles elementos que não estiverem ordenados. Basicamente, a cada passagem completa pelo conjunto, o item com o maior índice é posto no lugar correto. Lendo o livro do Iverson (1962), o temo *bubbling*, borbulhando, é usado como metáfora para indicar o que ocorre com cada item do conjunto, borbulhando até a sua posição. O Pseudocode 1, apresenta uma versão do Bubble Sort.

Pseudocode 1: Bubble Sort

```
for i = 1 to n - 1
    for j = 1 to n - i
        if (a[j] > a[j + 1]) then
            swap(a[j], a[j+1]
```

Consideraremos a letra i para indicar cada passada do algoritmo pelo conjunto que está sendo ordenado. Vamos considerar o melhor e o pior caso. No caso do Bubble sorte, independente da entrada, o número de passagens completas será dado por $n-1$. E para cada passagem o algoritmo irá realizar $n-i$ comparações. No melhor caso, o conjunto de entrada já está na ordem certa, nenhuma troca é realizada, ainda assim, passamos por todo conjunto $n-1$ vezes e verificamos a necessidade de troca, ou não, $n-i$ vezes. Ou seja, $n-1$ comparações serão feitas na primeira passagem, $n-2$ comparações na segunda passagem, $n-3$ comparações na terceira passagem e assim, sucessivamente até o fim do conjunto. Ou seja, o número total de passagens será:

$$(n-1)+(n-2)+(n-3)\ldots+3+2+1$$

Resolvendo esta soma temos:

$$\frac{n(n-1)}{2} \therefore \frac{n^2}{2}-\frac{n}{2}$$

Como podemos nos concentrar apenas no termo de maior ordem, temos a função assintótica dada por $O(n^2)$, para o Bubble Sort, nos três pontos de análise: pior cenário, os itens perfeitamente ordenados na ordem inversa da desejada; melhor cenário, o itens perfeitamente ordenados na ordem desejada e cenário médio, itens aleatoriamente distribuídos. A Implementação 1 apresenta uma versão do Bubble Sort em C++.

Implementação 1 Bubble Sort Tradicional.

```
/*
AUTHOR: Frank de Alcantara
DATA: 31 jul. 2020
Programa de demonstração do uso do Bubble Sort.
*/
#include <iostream>
#include <ctime>

using namespace std; // usando a biblioteca padrão

int main(){
    //iniciando o gerador randômico
    srand((unsigned)time(0));

    int temp = 0, tam = 10, passo = 0, conjunto[10];
    //preenchendo o conjunto (array a) com números randômicos.
    for (int i = 0; i < tam; i++){
        conjunto[i] = (rand() % 100) + 1; //randômicos < que cem
    }

    //imprime a lista criada que será ordenada só para testes
    cout << "\n\nLista original: {";
    for (int k = 0; k < tam; k++){
```

```
        cout << conjunto[k] << ",";
    }
    cout << "}" << endl;

    //Aqui está o Bubble Sort
    for (int i = 0; i < tam; i++){
        for (int j = i + 1; j < tam; j++){
            if (conjunto[j] < conjunto[i]){
                temp = conjunto[i]; //para fazer a troca
                conjunto[i] = conjunto[j];
                conjunto[j] = temp;
            }
        }
        passo++; //não faz parte do algoritmo. Conta as passagens
    }
    //imprimindo a lista ordenada
    cout << "\n\nLista ordenada: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
    }
    //imprimindo o total de passagens
    cout << "\nPassos: " << passo << endl;
}
```

Nesta implementação geramos um conjunto de itens aleatórios, populamos o *array* $conjunto[10]$ com estes números aleatórios, mostramos esta lista na tela, aplicamos o Bubble Sort sobre este *array*, mostramos no terminal o conjunto ordenado e, com uma pequena alteração, a inclusão da variável $passo$ podemos contar quantos passos são necessários para ordenar todo o conjunto.

O acréscimo de uma variável de monitoramento pode otimizar o algoritmo. Podemos usar esta variável no laço interno, dentro do *if* para indicar se em cada passagem alguma troca de posição foi realizada ou não. Caso uma passagem termine sem essa troca, encerramos o laço for externo. Este encerramento é feito com o comando *brake,* colocado em um if no fim do laço externo. A Implementação 2 mostra uma versão do Bubble Sort com essa alteração.

Implementação 2 – Bubble Sort Otimizado

```
/*
AUTHOR: Frank de Alcantara
DATA: 31 jul. 2020
Programa de demonstração do uso do Bubble Sort.
*/
#include <iostream>
#include <ctime>

using namespace std; // usando a biblioteca padrão

int main(){
    //iniciando o gerador randômicos
    srand((unsigned)time(0));

    int temp = 0, tam = 10, passo = 0, conjunto[10], monitora = 0;
    //preenchendo o conjunto (array a) com números randômicos.
    for (int i = 0; i < tam; i++){
        conjunto[i] = (rand() % 100) + 1; //randômicos < que cem
    }

    //imprime a lista criada que será ordenada só para testes
    cout << "\n\nLista original: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
    }
    cout << "}" << endl;

    //Aqui está o Bubble Sort
    for (int i = 0; i < tam; i++){
        for (int j = i + 1; j < tam; j++){
            if (conjunto[j] < conjunto[i]){
                temp = conjunto[i]; //para fazer a troca
                conjunto[i] = conjunto[j];
                conjunto[j] = temp;
                monitora = 1;
            }
```

```
        }
        passo++; //não faz parte do algoritmo. Conta as passagens
       if (monitora == 0){
     break; //encerra o laço principal se não haver trocas
    }
    //imprimindo a lista ordenada
    cout << "\n\nLista ordenada: {";
    for (int k = 0; k < tam; k++){
        cout << conjunto[k] << ",";
    }
    //imprimindo o total de passagens
    cout << "\nPassos: " << passo << endl;
}
```

Esta otimização economiza alguns ciclos de máquina já que o algoritmo irá parar assim que todos os itens estejam em posição e traz o melhor caso para a complexidade $O(n)$, mas não é de grande ajuda já que, no melhor caso, os itens já estão na posição desejada. SHAFFER, 2011 CORMEN, 2013 SEDGEWICK e FLAJOLET, 2013 SKIENA, 2008

3 REFERÊNCIAS

CORMEN, T. H. **Algorithms Unlocked**. Cambridege, MA. USA: Massachusetts Institute of Technology, 2013.

IVERSON, K. E. **A Programming Language**. [S.l.]: John Wiley, 1962.

KNUTH, D. E. Big Omicron and Big Omega and Big Theta. **SIGACT News**, p. 18-24, 1976.

SEDGEWICK, R.; FLAJOLET, P. **An Introduction to the Analysis of Algorithms**. Boston, MA. USA: Pearson Education, 2013.

SHAFFER, C. A. **Data Structures and Algorithm Analysis**. 3º. ed. Blacsburg, VA. USA: Dover Publications, 2011.

SKIENA, S. S. **The Algorithm Design Manual**. 2º. ed. London, UK: Springer-Verlag, 2008.