Quick Sort: Um Algoritmo Eficiente de Ordenação

Marco Aurelio Pereira Silva

Instituto Federal de Educação, Ciência e Tecnologia da Bahia – Campus Camaçari
CEP - 42808-590 – Camaçari – BA – Brasil

marcozsilva17@gmail.com

Resumo

O Quick Sort é amplamente utilizado por sua eficiência e simplicidade. Este artigo explora sua implementação, complexidade de tempo e espaço, e discute aplicações práticas. Também comparamos suas vantagens e desvantagens frente a outros algoritmos de ordenação. Organizar dados é um problema recorrente na Computação, pois muitas aplicações possuem uma grande quantidade de dados e a tendência, se os dados não forem previamente organizados, é a de que seja necessário um grande esforço computacional para encontrar os dados desejados. A fim de agilizar a procura e obter um bom desempenho foram criados os algoritmos de ordenação. Existe uma grande variedade destes algoritmos, tais como Bubble-sort, Merge-sort, Selection-sort e Insertion-Sort. No entanto, neste artigo, iremos tratar sobre o algoritmo Quick-Sort, por este ser um dos mais eficientes (em termos de tempo de ordenamento) em meio ao rol de algoritmos já propostos. Existem muitas variações do algoritmo Quick-Sort, a mais simples é o single-pivot, onde a ordenação ocorre com a utilização de apenas um pivô (elemento usado para auxiliar no processo de ordenamento). Com o tempo, outras implementações baseadas nesta solução mais simples de Quick-sort também se tornaram disponíveis, tais como o Quick-sort dual-pivot e multi-pivot. O algoritmo Quick-sort é conhecido como um dos mais rápidos, porém, estudos demonstraram que um dual-pivot tende a ordenar mais rapidamente um conjunto de dados, sugerindo que o aumento na de pivôs contribui para um melhor desempenho do algoritmo clássico Quick-sort.

O código fonte em que este artigo é baseado esta hospedado em:
https://github.com/MarquinhoZba/AnaliseAlgoritmo/blob/main/QuickSort.java

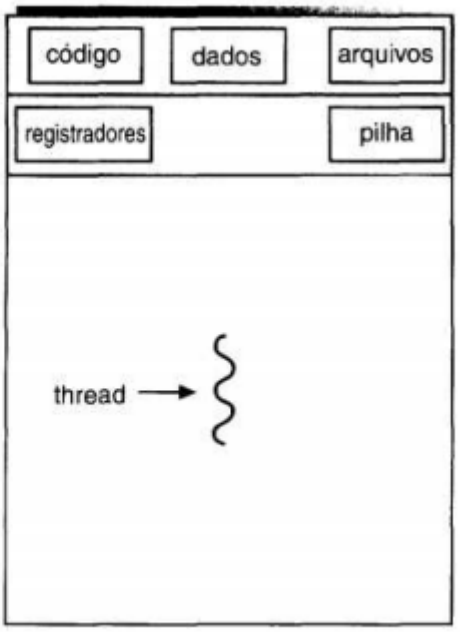
1. Introdução

A ordenação de dados é uma tarefa fundamental em ciência da computação e é aplicada em diversos contextos, como organização de dados em bases de dados, análise de dados e em algoritmos de busca. O Quick Sort, desenvolvido por Tony Hoare em 1960, é um dos algoritmos de ordenação mais populares. Sua abordagem de divisão e conquista permite ordenar elementos de maneira eficiente, o que o torna uma escolha preferencial para muitos programadores e engenheiros de software. Existe uma grande variedade destes algoritmos, tais como Bubble-sort, Merge-sort, Selection-sort e Insertion-Sort. No entanto, neste artigo, iremos tratar sobre o algoritmo Quick-Sort, por este ser um dos mais eficientes (em termos de tempo de ordenamento) em meio ao rol de algoritmos já propostos. Existem muitas variações

do algoritmo Quick-Sort, a mais simples é o single-pivot, onde a ordenação ocorre com a utilização de apenas um pivô (elemento usado para auxiliar no processo de ordenamento). Com o tempo, outras implementações baseadas nesta solução mais simples de Quick-sort também se tornaram disponíveis, tais como o Quick-sort dual-pivot e multi-pivot [AUMÜLLER et al., 2016]. O algoritmo Quick-sort é conhecido como um dos mais rápidos, porém, estudos demonstraram que um dual-pivot tende a ordenar mais rapidamente um conjunto de dados [FEOFILOFF, 2015], sugerindo que o aumento na de pivôs contribui para um melhor desempenho do algoritmo clássico Quick-sort.

3. THREAD

Quando um programa que tem uma grande demanda de atividades, ele tente a ser mais rapidamente executado se forem utilizados mecanismos realizar suas tarefas simultaneamente. A execução simultânea de vários fluxos de software pertencentes a um processo é denominada de Multithread, conforme ilustra a Figura 4. As threads podem ser facilmente criadas e destruídas [SILBERSCHATZ et al., 2004].



Fonte: SILBERSCHATZ *et al.*,2004.

2. Funcionamento do Quick Sort

O Quick Sort é um algoritmo recursivo que funciona da seguinte maneira:

- Escolha de um Pivô: Um elemento do array é escolhido como pivô. O desempenho do Quick Sort pode variar significativamente dependendo da escolha do pivô.

- Particionamento: Os elementos do array são rearranjados de forma que todos os elementos menores que o pivô fiquem à sua esquerda e todos os elementos maiores fiquem à sua direita.

- Recursão: O algoritmo é aplicado recursivamente nas duas sub-listas resultantes (à esquerda e à direita do pivô).

2.1. Implementação do Quick Sort

A seguir, apresentamos uma implementação do Quick Sort em Java, que ilustra os conceitos mencionados:

Em Java

```
import java.util.Arrays;
import java.util.Scanner;

public class QuickSort {

    // Função para particionar o array
    private static int partition(int[] array, int low, int high) {
        int pivot = array[high]; // Pivô
        int i = (low - 1); // Índice do menor elemento

        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
                // Troca array[i] e array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        // Troca array[i + 1] e array[high] (ou pivô)
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }

    // Função principal do Quick Sort
    private static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    // Função de wrapper para facilitar a chamada
```

```
    public static void quickSortWrapper(int[] array) {
        quickSort(array, 0, array.length - 1);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Informe o tamanho do array: ");
        int n = scanner.nextInt(); // Lê o tamanho do array
        int[] array = new int[n];

        // Lê os elementos do array
        System.out.println("Informe os elementos do array:");
        for (int i = 0; i < n; i++) {
            array[i] = scanner.nextInt();
        }

        System.out.println("Array original: " + Arrays.toString(array));

        quickSortWrapper(array);

        System.out.println("Array ordenado: " + Arrays.toString(array));

        scanner.close(); // Fecha o scanner
    }
}
```

4. Análise do Algoritmo

➔ Função `partition`

Descrição: A função `partition` organiza o array em torno de um pivô. Elementos menores ou iguais ao pivô são movidos à esquerda, enquanto elementos maiores ficam à direita.

Complexidade:

- O loop principal percorre de `low` até `high`, comparando e trocando elementos.
- A troca de elementos, em média, ocorre de forma constante, ou seja, **O(1)**.

Complexidade total: O(n), onde `n` é o número de elementos entre `low` e `high`.

➔ Função `quickSort`

Descrição: A função `quickSort` chama recursivamente a função `partition` para ordenar o array. A cada chamada, o array é particionado em duas partes, e a ordenação continua em cada parte.

Complexidade:

- No melhor caso, o pivô divide o array em partes quase iguais, resultando em uma profundidade de recursão logarítmica (`log n`), com cada chamada processando `n` elementos.

- No pior caso (quando o array já está ordenado ou inversamente ordenado), a recursão tem profundidade linear (`n`), o que resulta em uma complexidade quadrática.

Complexidade no melhor caso: O(n log n)
Complexidade no pior caso: O(n²)

➔ Função `quickSortWrapper`

Descrição: Esta função é apenas um invólucro que chama a função `quickSort` para o array completo. Não há loops ou recursões adicionais, então sua complexidade é determinada pela função `quickSort`.

Complexidade: O(n log n) no melhor caso e **O(n²)** no pior caso.

➔ Função `main`

Descrição: A função `main` lida com a entrada do usuário, inicializa o array e chama o método `quickSortWrapper` para ordenar o array. Após isso, exibe o array ordenado.

Complexidade:

- A leitura dos `n` elementos do array tem complexidade **O(n)**.
- Chamar `quickSortWrapper` leva **O(n log n)** no melhor caso e **O(n²)** no pior caso.

Complexidade total da função `main`: O(n log n) no melhor caso, **O(n²)** no pior caso.

Resumo da complexidade:

- **Função `partition`: O(n)**
- **Função `quickSort`: O(n log n)** no melhor caso, **O(n²)** no pior caso
- **Função `quickSortWrapper`: O(n log n)** no melhor caso, **O(n²)** no pior caso
- **Função `main`: O(n log n)** no melhor caso, **O(n²)** no pior caso

4.1. Análise de Espaço

A complexidade de espaço do **Quick Sor**t é geralmente **(O(log n))** devido à pilha de chamadas da recursão. No pior caso, pode chegar a **(O(n))**, embora o algoritmo seja classificado como um algoritmo in-place, o que significa que ele não requer espaço adicional proporcional ao tamanho da entrada.

4.2. Comparação com Outros Algoritmos

Quando comparado a outros algoritmos de ordenação, como Merge Sort e Bubble Sort, o Quick Sort se destaca devido à sua eficiência média. O Merge Sort tem uma complexidade de tempo garantida de **(O(n log n))**, mas usa mais espaço, enquanto o Bubble Sort é ineficiente com complexidade de tempo **(O(n^2))**.

5. Aplicações Práticas do Quick Sort

O Quick Sort é utilizado em várias aplicações do mundo real, como:

- ➔ Sistemas de Banco de Dados: Para ordenar registros de forma eficiente.
- ➔ Análise de Dados: Em ferramentas de análise de dados que exigem ordenação rápida.
- ➔ Algoritmos de Busca: Para melhorar a eficiência na busca de dados.

6. Conclusão

O Quick Sort combina simplicidade e eficiência, sendo amplamente aplicado em diversos cenários. Mesmo com suas limitações no pior caso, sua capacidade de operar in-place e sua eficiência média o tornam indispensável para desenvolvedores e engenheiros de software. Por sua adaptabilidade e performance, o Quick Sort continua a ser uma ferramenta crucial no campo da ordenação de dados.

Referências

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

2. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

3. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.

4. Patel, M., & Aggarwal, R. (2017). *Advanced Sorting Algorithms: A Comparative Study of Dual-Pivot Quick Sort*. International Journal of Computer Applications.

5. Kaligari, A., & Singla, A. (2020). *Optimizing Quick Sort with Multi-Pivot Variations*. IEEE Access.