Conceitos de Algoritmos de Programação I

Lorenzo Calabrese Circelli

26 de maio de 2025

Resumo

Algoritmos são sequências de passos bem definidos que resolvem problemas computacionais, sendo a base lógica da programação. Entre os diversos tipos, os algoritmos de ordenação têm papel fundamental, pois organizam dados em ordem crescente ou decrescente, otimizando buscas, análises e visualizações. Entender esses métodos vai além de saber ordenar: envolve compreender estratégias, desempenho (tempo e espaço), estabilidade e aplicações práticas. Estudar técnicas como Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort desenvolve o raciocínio lógico e a intuição algorítmica essenciais para a construção de soluções eficientes.

1 Introdução

Na base de toda solução computacional está um algoritmo — uma sequência finita de passos bem definidos para resolver um problema. Seja para calcular, tomar decisões, percorrer dados ou transformá-los, os algoritmos são o alicerce lógico da programação. Dentre os diversos tipos de algoritmos, os métodos de ordenação ocupam um papel fundamental. Eles são responsáveis por organizar conjuntos de dados em uma determinada ordem, normalmente crescente ou decrescente, e são essenciais para melhorar a eficiência de buscas, análises e visualizações de informações.

Compreender os algoritmos de ordenação não é apenas uma questão de saber como ordenar uma lista — é entender a lógica por trás de cada estratégia, os critérios de desempenho (tempo, espaço, estabilidade) e a aplicabilidade em diferentes cenários. Através do estudo de métodos como Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, entre outros, é possível desenvolver intuição algorítmica e raciocínio lógico, habilidades essenciais para qualquer programador.

2 Nivelamento

2.1 Entender o que são algoritmos

Na programação, um algoritmo é uma sequência lógica, finita e bem definida de instruções que orientam o computador na execução de uma tarefa específica. Ele funciona como uma receita que descreve passo a passo como transformar dados de entrada em resultados desejados. Cada passo do algoritmo deve ser claro e preciso, para que a máquina possa interpretá-lo e executá-lo sem ambiguidade.

Os algoritmos são a base de qualquer programa de computador: todo software, por mais complexo que seja, é composto por algoritmos que determinam seu comportamento. Eles são utilizados para resolver problemas computacionais, automatizar processos, tomar decisões, ordenar dados, fazer buscas, entre muitas outras funções.

Na prática da programação, um algoritmo pode ser implementado em qualquer linguagem (como C, Java, Python, etc.), mas sua lógica é independente da linguagem usada. Ao aprender algoritmos, o foco está em desenvolver a capacidade de pensar logicamente, estruturar soluções eficientes e aplicar estratégias adequadas para diferentes tipos de problema.

Além disso, é importante considerar características como eficiência (quanto tempo e memória o algoritmo consome), corretude (se ele sempre produz o resultado certo) e legibilidade (se ele é fácil de entender e manter). Por isso, o estudo de algoritmos é essencial para formar uma base sólida em programação e preparar o desenvolvedor para enfrentar desafios reais de desenvolvimento de software.

2.2 O que é a notação Big O?

A **notação Big O** é uma forma de descrever o comportamento de um algoritmo em relação ao seu desempenho, especialmente quando lidamos com grandes volumes de dados. Ela representa, de maneira simplificada, *como o tempo de execução ou o uso de memória de um algoritmo cresce à medida que o tamanho da entrada aumenta.*

Diferente de medir tempos exatos de execução, a notação Big O se concentra na **ordem de crescimento**, ou seja, como o algoritmo escala com o aumento do número de elementos. Isso é essencial para comparar a eficiência entre algoritmos e fazer escolhas adequadas para diferentes contextos.

Alguns exemplos comuns de notação Big O:

- $\mathcal{O}(1)$: tempo constante – o tempo de execução não depende do tamanho da entrada.
- $\mathcal{O}(n)$: tempo linear – o tempo cresce proporcionalmente ao número de elementos.
- $\mathcal{O}(n^2)$: tempo quadrático – o tempo cresce com o quadrado do número de elementos (comum em algoritmos como o *Bubble Sort*).
- $\mathcal{O}(\log n)$: tempo logarítmico – muito eficiente; o tempo cresce lentamente mesmo com grandes entradas.
- $\mathcal{O}(n \log n)$: encontrado em algoritmos de ordenação eficientes como *Merge Sort* e *Quick Sort*.
- $\mathcal{O}(n!)$: tempo fatorial – cresce extremamente rápido, tornando algoritmos com essa complexidade inviáveis para entradas grandes; comum em problemas de força bruta, como o problema do caixeiro viajante.

A notação Big O ignora constantes e termos menores, focando no **comportamento dominante** do algoritmo. Compreender essa notação é fundamental para analisar a **eficiência** de algoritmos, prever o desempenho de sistemas e desenvolver soluções escaláveis e otimizadas.

3 Algoritmos de Ordenação

Nesta seção, apresentamos os principais algoritmos de ordenação utilizados em programação, cada um com sua descrição, código em C e explicação do funcionamento.

3.1 Bubble Sort

O *Bubble Sort* é um dos algoritmos de ordenação mais simples. Ele funciona comparando pares de elementos adjacentes e trocando-os de lugar se estiverem na ordem errada, repetindo o processo até que a lista esteja ordenada.

```
void bubbleSort(int A[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - 1 - i; j++) {
            if (A[j] > A[j + 1]) {
                temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            }
        }
    }
}
```

Explicação: O laço externo controla o número total de passagens pela lista, que é $n-1$. A cada passagem, o laço interno percorre a lista até o elemento antes da última posição ordenada, que cresce a cada iteração, evitando reprocessar elementos já posicionados corretamente. Se um par adjacente estiver fora de ordem (o elemento da esquerda maior que o da direita), eles são trocados. Essa operação se repete até que todos os elementos estejam ordenados do menor para o maior. Por causa das múltiplas comparações e trocas, o algoritmo tem complexidade $O(n^2)$.

O algoritmo realiza várias passagens pela lista, em cada uma "borbulhando"o maior elemento não ordenado para o final. Isso é feito comparando pares adjacentes e trocando-os se necessário. O processo se repete até que nenhuma troca seja feita, indicando que a lista está ordenada.

3.2 Selection Sort

O *Selection Sort* ordena a lista repetidamente encontrando o menor elemento da parte não ordenada e trocando-o com o primeiro elemento dessa parte.

```
void selectionSort(int A[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (A[j] < A[minIndex]) {
                minIndex = j;
            }
        }
        temp = A[i];
        A[i] = A[minIndex];
        A[minIndex] = temp;
    }
}
```

Explicação: No laço externo, o índice i marca a posição onde o próximo menor elemento será colocado. O laço interno procura na parte não ordenada (a partir de $i+1$) o índice do menor elemento.

Após encontrar esse índice (`minIndex`), o valor é trocado com o elemento da posição i. Assim, a parte à esquerda de i fica ordenada e cresce a cada iteração. O algoritmo também possui complexidade $O(n^2)$, mas realiza menos trocas que o Bubble Sort.

O algoritmo divide a lista em duas partes: a ordenada e a não ordenada. A cada iteração, ele seleciona o menor elemento da parte não ordenada e o coloca na posição correta, expandindo a parte ordenada.

3.3 Insertion Sort

O *Insertion Sort* constrói a lista ordenada um elemento por vez, inserindo cada novo elemento na posição correta em relação aos anteriores já ordenados.

```
void insertionSort(int A[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = A[i];
        j = i - 1;
        while (j >= 0 && A[j] > key) {
            A[j + 1] = A[j];
            j--;
        }
        A[j + 1] = key;
    }
}
```

Explicação: Começando do segundo elemento ($i = 1$), o algoritmo armazena o valor atual em `key` e compara com os elementos anteriores. Enquanto os elementos anteriores forem maiores que `key`, eles são deslocados uma posição à direita para abrir espaço. Quando a posição correta for encontrada (onde $A[j] \leq key$ ou $j < 0$), `key` é inserido. Esse processo é repetido para cada elemento até que toda a lista esteja ordenada. O Insertion Sort é eficiente para listas quase ordenadas, com complexidade média e pior caso $O(n^2)$, mas em casos ótimos pode chegar a $O(n)$.

O algoritmo percorre a lista da esquerda para a direita, e em cada passo insere o elemento atual na posição correta, movendo os elementos maiores para a direita. É eficiente para listas pequenas ou quase ordenadas.

3.4 Merge Sort

O *Merge Sort* é um algoritmo do tipo "dividir para conquistar". Ele divide a lista em duas metades, ordena cada uma recursivamente e depois as combina (merge) em uma lista ordenada.

```
void merge(int A[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    int i, j, k;

    for (i = 0; i < n1; i++)
        L[i] = A[left + i];
    for (j = 0; j < n2; j++)
        R[j] = A[mid + 1 + j];

    i = 0; j = 0; k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            A[k++] = L[i++];
        } else {
            A[k++] = R[j++];
        }
    }

    while (i < n1) {
        A[k++] = L[i++];
    }
    while (j < n2) {
        A[k++] = R[j++];
    }
}

void mergeSort(int A[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(A, left, mid);
        mergeSort(A, mid + 1, right);
        merge(A, left, mid, right);
    }
}
```

Explicação: A função `mergeSort` divide a lista em duas metades enquanto o intervalo for maior que um elemento. Recursivamente, ordena as metades esquerda e direita. Após ordenar as sublistas, a função `merge` é chamada para fundi-las em ordem crescente. Essa função cria arrays temporários `L` e `R` para armazenar as metades, e usa três índices para comparar e inserir os elementos de forma ordenada no array original. O Merge Sort possui complexidade $O(n \log n)$ em todos os casos e é estável, porém utiliza espaço extra para os arrays temporários.

O Merge Sort divide a lista recursivamente até obter listas unitárias, que são naturalmente ordenadas, e depois as une em ordem crescente. Esse algoritmo tem boa performance, mesmo para listas grandes, com complexidade $\mathcal{O}(n \log n)$.

3.5 Quick Sort

O *Quick Sort* também utiliza o método "dividir para conquistar". Ele escolhe um elemento chamado pivô, particiona a lista em elementos menores e maiores que o pivô e aplica recursivamente o mesmo processo às sublistas.

```
int partition(int A[], int low, int high) {
    int pivot = A[high];
    int i = low - 1;
    int temp;
    for (int j = low; j < high; j++) {
        if (A[j] <= pivot) {
            i++;
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }
    temp = A[i + 1];
    A[i + 1] = A[high];
    A[high] = temp;
    return i + 1;
}

void quickSort(int A[], int low, int high) {
    if (low < high) {
        int pi = partition(A, low, high);
        quickSort(A, low, pi - 1);
        quickSort(A, pi + 1, high);
    }
}
```

Explicação: A função `partition` escolhe o último elemento como pivô. Percorre o array do índice `low` até `high-1`, movendo os elementos menores ou iguais ao pivô para a esquerda. O índice `i` indica a última posição do grupo dos menores. Após o laço, o pivô é colocado imediatamente após esse grupo, garantindo que todos à esquerda sejam menores e à direita maiores. A função retorna a posição do pivô. O `quickSort` chama-se recursivamente para as sublistas esquerda e direita, até ordenar completamente. A complexidade média é $O(n \log n)$, porém no pior caso pode ser $O(n^2)$, especialmente quando o pivô é mal escolhido.

O algoritmo reorganiza os elementos para que os menores que o pivô fiquem à esquerda e os maiores à direita. Depois, ordena recursivamente as duas partes. O Quick Sort é rápido na prática e bastante eficiente para a maioria dos casos, sendo amplamente utilizado em aplicações reais devido à sua boa performance e simplicidade relativa.

4 Dica de Ouro - LeetCode

Atualmente, essa é a realidade do jogo: se você souber como encarar o LeetCode, vai conquistar uma excelente oferta em uma empresa top. É simples assim.

Veja como usar o LeetCode da maneira correta:

NÃO tente resolver nenhum problema sozinho (pelo menos ainda não!). Pode parecer estranho, mas realmente é assim. Gastar tempo tentando resolver qualquer questão sem a base adequada é inútil. Nem gaste 5 minutos tentando, você vai acabar frustrado e sem progresso.

Então, qual é o caminho? É bem simples.

Comece pelo *Grokking the Coding Interview* (mas não compre, é um gasto desnecessário) e veja a lista de padrões que eles indicam.

Escolha um padrão e vá para o LeetCode. Procure problemas que seguem esse padrão.

Para cada problema, vá **DIRETAMENTE para a solução**. Não perca tempo tentando resolver antes, isso é perda de tempo.

Estude a solução com profundidade. Anote tudo. Se algo não ficar claro, pesquise, assista vídeos no YouTube sobre o problema. Confira a seção de discussões no LeetCode para ver como outras pessoas resolveram. Teste a solução, mexa nas variáveis, experimente. O importante é entender a solução profundamente.

Depois, passe para o próximo problema e repita o processo.

Depois de trabalhar com vários problemas assim, você vai perceber que tudo começa a fazer sentido. Parabéns, agora você sabe resolver esse padrão!

Escolha outro padrão e faça o mesmo.

Como você não está gastando horas tentando resolver cada problema, em apenas 1 a 3 semanas terá um entendimento sólido dos principais padrões e das soluções mais comuns. Você será capaz de identificar como dividir problemas complexos em padrões conhecidos.

Após passar por 300 a 400 problemas dessa forma (parece muito, mas lembre-se, você não está perdendo horas tentando resolver, então consegue avançar rápido), você estará pronto para encarar questões específicas de empresas. Para isso, pode adquirir o LeetCode Premium e verá que agora consegue resolver esses desafios sozinho!

Parabéns, você acaba de economizar meses e meses de frustração.

Fonte: Adaptado do Reddit, subreddit r/brdev, usuário dinizzdev

5 Recomendações de Fontes de Estudo

Para aprofundar no estudo dos algoritmos e entender melhor, com explicações mais detalhadas e de fontes respeitadas no cenário, muitas das quais contribuíram ativamente para termos o nível de entendimento e desenvolvimento que possuímos hoje, aqui vão algumas dicas e recomendações pessoais, que eu considero importantes de se ter pelo menos na manga.

5.1 Livros Clássicos e Referências Fundamentais

- **Introduction to Algorithms** — Cormen, Leiserson, Rivest e Stein (CLRS).
  Considerado um dos livros mais completos sobre algoritmos, apresenta fundamentos teóricos, técnicas de projeto e análise detalhada de diversos algoritmos, incluindo métodos de ordenação.
- **The Art of Computer Programming** — Donald Knuth.
  Obra monumental que explora algoritmos com profundidade matemática. O Volume 3 é dedicado a algoritmos de ordenação e pesquisa, ideal para estudo aprofundado.
- **Algorithms** — Robert Sedgewick e Kevin Wayne.
  Texto didático e moderno com enfoque prático e visualização, acompanhado de exemplos em Java. Possui também cursos online disponíveis.
- **Algorithm Design** — Jon Kleinberg e Éva Tardos.
  Excelente para aprender técnicas de design de algoritmos, com capítulos dedicados a algoritmos de ordenação e análise de desempenho.

5.2 Artigos e Publicações Importantes

- **A New Sorting Algorithm** — C. A. R. Hoare (1962).
  Artigo original que introduz o algoritmo Quick Sort, fundamental para compreender sua motivação e funcionamento.
- **Artigos sobre otimizações e variantes de algoritmos de ordenação.**
  Pesquisas disponíveis em bases acadêmicas como Google Scholar e periódicos especializados, que discutem melhorias e variações modernas.
- **Survey papers em periódicos como ACM Computing Surveys.**
  Revisões anuais sobre avanços em algoritmos e suas aplicações.

5.3 Recursos Online Dinâmicos e Interativos

- **VisuAlgo** – https://visualgo.net/pt.
  Plataforma para visualização passo a passo de algoritmos, incluindo ordenação.
- **GeeksforGeeks** – https://www.geeksforgeeks.org.
  Biblioteca online com explicações, exemplos de código e desafios práticos.
- **Cursos em Coursera e edX**.
  Cursos ministrados por universidades renomadas, com aulas, quizzes e projetos práticos.
- **Canais no YouTube**.
  Exemplos: *mycodeschool*, *Abdul Bari*, *Tushar Roy* — com explicações visuais e didáticas.

5.4 Comunidades e Fóruns

- **Stack Overflow e Stack Exchange (Computer Science)**.
  Para tirar dúvidas práticas e discutir detalhes específicos.
- **Reddit – r/algorithms**.
  Discussões, notícias e troca de conhecimentos na área de algoritmos.
- **GitHub**.
  Repositórios públicos com implementações diversas para estudo e prática.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.

[2] Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th Edition, Addison-Wesley Professional, 2011.

[3] Gayle Laakmann McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*, 6th Edition, CareerCup, 2015.