Estruturas de Dados

Análise de Algoritmos

Prof. Igor de Moraes Sampaio
igor.sampaio@ifsp.edu.br

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - Campus Campos do Jordão

Análise de Algoritmos

A Análise de Algoritmos é um campo fundamental da Ciência da Computação que estuda a eficiência dos algoritmos em termos de tempo de execução e uso de memória. Essa análise permite escolher o algoritmo mais adequado para resolver um problema, garantindo desempenho otimizado

O que é Análise de Algoritmos?

- A análise de algoritmos busca determinar a eficiência de um algoritmo ao processar entradas de diferentes tamanhos. Seu principal objetivo é prever a escalabilidade e o impacto dos recursos computacionais.

- Ela envolve:

    - **Complexidade de tempo:** Quanto tempo o algoritmo leva para processar a entrada.

    - **Complexidade de espaço:** Quantidade de memória necessária para executar o algoritmo.

Importância da Análise de Algoritmos

- Escolher o algoritmo mais eficiente para um problema.
- Garantir que o programa pode lidar com grandes quantidades de dados.
- Melhorar a utilização dos recursos computacionais.
- Permitir previsibilidade e comparação entre diferentes abordagens.

Tipos de Complexidade

A complexidade é dividida em dois aspectos principais:

- **Complexidade de Tempo**
  - Mede a quantidade de operações executadas pelo algoritmo conforme o tamanho da entrada cresce.

- **Complexidade de Espaço**
  - Mede a quantidade de memória extra necessária para armazenar variáveis, pilhas de recursão e estruturas auxiliares.

Notação Assintótica

O - Notation (Big-O):

- Usada para expressar o pior caso de um algoritmo. Representa um limite superior do tempo de execução.

- $\mathcal{O}(1) \rightarrow$ Constante
- $\mathcal{O}(\log n) \rightarrow$ Logarítmica
- $\mathcal{O}(n) \rightarrow$ Linear
- $\mathcal{O}(n \log n) \rightarrow$ Quasilinear
- $\mathcal{O}(n^2) \rightarrow$ Quadrática
- $\mathcal{O}(2^n) \rightarrow$ Exponencial
- $\mathcal{O}(n!) \rightarrow$ Fatorial

Ω - Notation (Big-Omega)

- Representa o melhor caso do algoritmo.

Θ - Notation (Big-Theta)

- Representa o comportamento médio do algoritmo.

Exemplos de Análise de Complexidade

Busca Linear - $\mathcal{O}(n)$

Percorre todos os elementos de uma lista até encontrar o valor desejado.

Análise: No pior caso, a busca percorre toda a lista → $\mathcal{O}(n)$

```c
int buscaLinear(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            // Retorna o índice do elemento encontrado
            return i;
        }
    }
    // Retorna -1 se o elemento não for encontrado
    return -1;
}
```

Busca Binária - $\mathcal{O}(\log n)$

Utiliza um vetor ordenado e divide-o ao meio a cada passo.

Análise: A cada iteração reduzimos o problema pela metade, resultando em $\mathcal{O}(\log n)$

```c
int buscaBinaria(int arr[], int n, int x) {
    int inicio = 0, fim = n - 1;

    while (inicio <= fim) {
        int meio = inicio + (fim - inicio) / 2;

        if (arr[meio] == x)
            return meio;
        else if (arr[meio] < x)
            inicio = meio + 1;
        else
            fim = meio - 1;
    }
    return -1;
}
```

Bubble Sort - $\mathcal{O}(n^2)$

Um algoritmo de ordenação ineficiente que compara e troca elementos vizinhos.

Análise: Para cada elemento, percorremos todos os outros → $\mathcal{O}(n^2)$

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

Merge Sort - $\mathcal{O}(n \log n)$

Divide a lista ao meio e ordena recursivamente.

Análise: Divide o problema em partes menores e junta os resultados → $\mathcal{O}(n \log n)$

```c
void mergeSort(int arr[], int esquerda, int direita) {
    if (esquerda < direita) {
        int meio = esquerda + (direita - esquerda) / 2;

        mergeSort(arr, esquerda, meio);
        mergeSort(arr, meio + 1, direita);

        merge(arr, esquerda, meio, direita);
    }
}
```

```c
void merge(int arr[], int esquerda, int meio, int direita) {
    int n1 = meio - esquerda + 1;
    int n2 = direita - meio;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[esquerda + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[meio + 1 + j];

    int i = 0, j = 0, k = esquerda;

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

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Análise Empírica vs. Teórica

- A análise de algoritmos pode ser realizada de duas formas:
    - **Análise Teórica:** Usa expressões matemáticas para determinar a complexidade.
    - **Análise Empírica:** Mede o tempo de execução real rodando o algoritmo em diferentes entradas.

Algoritmo	Melhor Caso	Médio Caso	Pior Caso
Busca Linear	$O(1)$	$O(n)$	$O(n)$
Busca Binária	$O(1)$	$O(\log n)$	$O(\log n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Comparação de Algoritmos

Estratégias de Otimização

- **Usar estruturas de dados eficientes:** Exemplo, tabelas hash para buscas rápidas.
- **Escolher algoritmos adequados:** Usar Merge Sort ao invés de Bubble Sort.
- **Evitar cálculos desnecessários:** Memorizar resultados de cálculos repetitivos.
- **Utilizar técnicas de programação dinâmica:** Para reduzir complexidade exponencial.

Conclusão

A análise de algoritmos é essencial para o desenvolvimento de sistemas eficientes. Escolher o melhor algoritmo pode economizar tempo e recursos computacionais, tornando os sistemas escaláveis e otimizados.