Algoritmos e Estruturas de dados

1. Algoritmos

Algoritmo é uma sequência finita de passos ou instruções bem definidos para resolver um problema ou realizar uma tarefa específica. A eficiência de um algoritmo é avaliada principalmente pela sua complexidade de tempo e espaço.

a) Complexidade de Algoritmos

A complexidade de um algoritmo mede a quantidade de recursos que ele consome, como tempo de execução e memória, em função do tamanho da entrada. Há dois principais tipos de complexidade:

- **Complexidade de Tempo:** Mede o tempo necessário para a execução do algoritmo. É expressa normalmente em termos de notação **O(grande)**, que representa o comportamento assintótico do algoritmo.
    - Exemplos de notações de complexidade:
        - O(1): Tempo constante.
        - O(log n): Tempo logarítmico.
        - O(n): Tempo linear.
        - $O(n^2)$: Tempo quadrático.
        - O(2^n): Tempo exponencial.
- **Complexidade de Espaço:** Mede a quantidade de memória que o algoritmo utiliza.

b) Tipos de Algoritmos

Os algoritmos podem ser classificados de diversas formas, com base em seu paradigma de resolução. Alguns dos paradigmas mais importantes são:

- **Dividir e Conquistar:** Divide o problema em subproblemas menores e resolve cada um de forma recursiva. Exemplos: algoritmo de ordenação Merge Sort, Quick Sort.
- **Algoritmos Gulosos:** Tomam decisões baseadas na escolha local mais "gula" ou promissora, sem considerar o problema completo. Exemplos: Algoritmo de Kruskal, Algoritmo de Dijkstra.
- **Programação Dinâmica:** Resolve subproblemas menores e armazena os resultados para evitar recomputação. Exemplos: Fibonacci, Algoritmo de Bellman-Ford.

- **Força Bruta:** Avalia todas as soluções possíveis para encontrar a solução ótima. Geralmente tem uma complexidade alta, mas é garantido que encontra a solução.
- **Backtracking:** Explora todas as possíveis soluções através de uma abordagem de tentativa e erro, mas retrocede quando uma solução falha. Exemplo: Resolução de Sudoku.

2. Estruturas de Dados

Estruturas de dados são maneiras de organizar e armazenar dados em um computador para que possam ser acessados e modificados de maneira eficiente. As estruturas de dados determinam a eficiência das operações de busca, inserção e remoção de elementos.

a) Listas

- **Lista Simplesmente Encadeada:** Cada elemento (ou nó) contém um valor e uma referência para o próximo elemento. A inserção e a remoção em uma lista encadeada podem ser feitas de forma eficiente, mas o acesso a um elemento é mais lento do que em arrays.
- **Lista Duplamente Encadeada:** Cada nó tem referências para o próximo e o anterior, o que facilita a navegação bidirecional.

b) Pilhas (Stacks)

- Funcionam no princípio **LIFO (Last In, First Out)**, ou seja, o último elemento inserido é o primeiro a ser removido. Operações principais:
    - **Push:** Inserir um elemento no topo da pilha.
    - **Pop:** Remover o elemento do topo da pilha.
    - **Peek:** Consultar o elemento no topo sem removê-lo.
- Usos comuns: avaliação de expressões, navegação em páginas da web (voltar e avançar).

c) Filas (Queues)

- Funcionam no princípio **FIFO (First In, First Out)**, onde o primeiro elemento inserido é o primeiro a ser removido. Operações principais:
    - **Enqueue:** Inserir um elemento no final da fila.
    - **Dequeue:** Remover o elemento no início da fila.
- Variantes:

    - **Fila Circular:** Usa um array de tamanho fixo, onde o último elemento "envolve" para o início.
    - **Fila de Prioridade:** Cada elemento tem uma prioridade associada, e os elementos com maior prioridade são removidos primeiro.

d) Árvores

Árvores são estruturas de dados hierárquicas, onde cada elemento (nó) tem um valor e pode ter "filhos". O nó mais alto é chamado de **raiz**, e os nós sem filhos são chamados de **folhas**.

- **Árvore Binária:** Cada nó tem no máximo dois filhos (esquerdo e direito). Em árvores binárias, a busca, inserção e remoção de elementos pode ser feita de forma eficiente, especialmente em **árvores binárias de busca**.
- **Árvore Binária de Busca (BST):** Uma árvore binária onde os nós à esquerda de um nó contêm valores menores, e os nós à direita contêm valores maiores. Permite operações eficientes de busca (em O(log n) em média, se a árvore for balanceada).
- **Árvore AVL e Árvore Red-Black:** São exemplos de **árvores balanceadas**, que garantem um tempo de busca O(log n), ajustando automaticamente a profundidade para evitar o pior caso de desempenho.
- **Heap:** Uma árvore binária completa onde cada nó é maior (max-heap) ou menor (min-heap) que seus filhos. Usada principalmente para implementar filas de prioridade.

e) Grafos

- **Grafo:** Consiste em um conjunto de nós (ou vértices) e conexões (arestas) entre eles. Grafos podem ser **direcionados** ou **não-direcionados**, e podem ter **pesos** associados às arestas.
- **Algoritmos de Grafos:**
    - **Busca em Largura (BFS):** Explora o grafo nível por nível, começando por um nó de origem.
    - **Busca em Profundidade (DFS):** Explora o grafo seguindo cada caminho até o fim antes de voltar.
    - **Dijkstra:** Algoritmo para encontrar o caminho mais curto em grafos com arestas de peso positivo.
    - **Kruskal e Prim:** Algoritmos para encontrar a árvore geradora mínima em um grafo.

3. Métodos de Acesso, Busca, Inserção e Ordenação

a) Métodos de Busca

- **Busca Linear (O(n)):** Verifica cada elemento da estrutura de dados até encontrar o valor desejado.
- **Busca Binária (O(log n)):** Requer uma estrutura ordenada. Divide a lista ao meio repetidamente até encontrar o elemento.

b) Métodos de Ordenação

- **Bubble Sort (O(n^2)):** Comparação repetida de elementos adjacentes e troca de posições se estiverem fora de ordem.
- **Selection Sort (O(n^2)):** Seleciona o menor elemento e coloca-o na posição correta repetidamente.
- **Insertion Sort (O(n^2)):** Insere elementos de uma lista desordenada na posição correta de uma lista ordenada.
- **Merge Sort (O(n log n)):** Divide a lista em sublistas até que cada uma tenha um único elemento, e então as combina de volta em ordem.
- **Quick Sort (O(n log n) em média):** Escolhe um elemento como pivô e particiona a lista em duas partes; a primeira com elementos menores que o pivô, e a segunda com elementos maiores.