Passos para Analisar a Complexidade Big O

1. **Entender o Algoritmo**: Analise o código para identificar quais operações são realizadas e como elas são afetadas pelo tamanho da entrada.
2. **Identificar Operações Relevantes**: Determine as operações mais importantes que influenciam o tempo de execução, como verificações de condição, inserções e remoções.
3. **Determinar o Tempo de Execução**: Avalie quantas dessas operações são realizadas em relação ao tamanho da entrada.

1. O(1) - Complexidade Constante

- **Significado**: `O(1)` indica que o tempo de execução ou o uso de memória do algoritmo é constante e não depende do tamanho da entrada. Em outras palavras, o algoritmo realiza uma quantidade fixa de trabalho independentemente de quantos dados ele está processando.
- **Exemplo**:
    - Acesso a um elemento em um array por índice (e.g., `array[index]`).
    - Operação `push` ou `pop` em uma pilha (como discutido anteriormente).

Por que é importante?: Algoritmos com complexidade `O(1)` são muito eficientes, pois a execução não se torna mais lenta com o aumento do tamanho dos dados.

2. O(n) - Complexidade Linear

- **Significado**: `O(n)` indica que o tempo de execução ou o uso de memória do algoritmo cresce linearmente com o tamanho da entrada. Se o tamanho da entrada dobrar, o tempo de execução ou o uso de memória também dobrará.
- **Exemplo**:
    - Percorrer todos os elementos de uma lista para realizar uma operação (e.g., somar todos os elementos de uma lista).
    - Um loop simples que itera através de um array de n elementos.

Por que é importante?: Algoritmos com complexidade `O(n)` são eficientes para entradas de tamanho moderado, mas podem se tornar lentos com dados muito grandes.

3. O(m) - Complexidade Linear em Relação a Outro Parâmetro

- **Significado**: `O(m)` é semelhante a `O(n)`, mas refere-se a uma variável diferente (`m`). A complexidade `O(m)` indica que o tempo de execução ou o uso de memória cresce linearmente com o tamanho do parâmetro `m`.
- **Exemplo**:
  - Processar uma lista de strings onde o número de strings é `m`.
  - Um algoritmo que percorre uma matriz de dimensões `m x n`, onde o tempo de execução é proporcional ao número de linhas `m`.

Por que é importante?: A notação `O(m)` é útil quando o desempenho do algoritmo depende de múltiplos parâmetros, e ajuda a entender como o algoritmo se comporta em relação a cada um desses parâmetros.

Comparação entre Notações

- **`O(1)`** é o melhor caso em termos de eficiência, pois o tempo de execução ou o uso de memória não cresce com o tamanho da entrada.
- **`O(n)`** é eficiente, mas o tempo de execução ou o uso de memória cresce linearmente com o tamanho da entrada.
- **`O(m)`** é útil para descrever algoritmos que têm complexidade linear em relação a um parâmetro específico, diferente do tamanho total da entrada.

Exemplos Práticos

4. **Acesso a um Array**:
   - Código: `int value = array[index];`
   - Complexidade: `O(1)`
5. **Soma de Todos os Elementos de um Array**:
   - Código: `int sum = 0; for (int i = 0; i < array.length; i++) { sum += array[i]; }`
   - Complexidade: `O(n)`, onde n é o número de elementos no array.
6. **Processamento de Uma Lista de Strings**:
   - Código: `for (String s : list) { process(s); }`
   - Complexidade: `O(m)`, onde `m` é o número de strings na lista.

Resumo

- **`O(1)`**: Tempo ou espaço constante, não depende do tamanho da entrada.
- **`O(n)`**: Tempo ou espaço linear, cresce proporcionalmente ao tamanho da entrada.
- **`O(m)`**: Tempo ou espaço linear em relação a um parâmetro específico `m`.