Análise de Complexidade de Algoritmo

Enzo Tiriba Appi Matricula: 1801130023

Complexidade Constante: O(1)

As operações a seguir seguem *complexidade constante O(1)*:

- **Atribuição de valor à variavel.**
- **Inserção de elemento em *array.***
- **Determinar se um número é par ou ímpar.**
- **Recuperar um elemento (i) de um a*rray.***
- **Recuperar um valor de uma *hash table*(dicionário) com uma *key.***

Essas operações são ditas de *complexidade constante*, pois são declarações simples realizadas em cada elemento de entrada.

Exemplo de trecho de código:

```
void imprimirElementoVetor(int vetor[]) {

    printf("Primeiro elemento do vetor = %d", vetor[0]);

}
```

Como você pode ver no gráfico abaixo o tempo constante é indiferente ao tamanho da entrada. Declarando uma variável, inserindo um elemento em uma pilha, inserindo um elemento em uma lista vinculada não classificada todas essas declarações levam tempo constante.

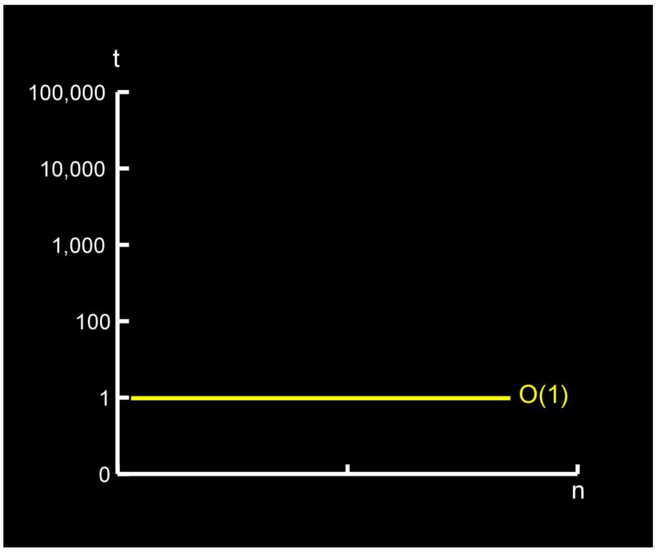

Figura 1: Gráfico de complexidade de algoritmo. Tempo de execução (eixo y) x numero de iteraçoes n (eixo x).

Complexidade Linear: O(n)

As operações a seguir seguem *complexidade linear* O(n):

- **Encontrar um item em uma coleção não classificada ou uma árvore desequilibrada (pior caso).**
- **Classificando uma matriz através de tipo bolha.**

Exemplo de trecho de código:

```
void imprimeTodosElementosVetor(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("%d\n", vetor[i]);
    }
}
```

No trecho de código decscrito, um laço (*loop)* executa iterativamente, n vezes, uma operação declarada em seu interior. Observa-se que a operação declarada no escopo do referido laço, de complexidade O(1), é realizada n vezes (i, nesse caso). Portanto, afirma-se que o tempo para a execução total do laço (*loop)* é N * O(1); o que equivale a dizer que o tempo é O(N), também, conhecido como *complexidade linear*:

No gráfico a seguir, podemos ver como o tempo de execução aumenta linearmente em relação ao número de elementos n:

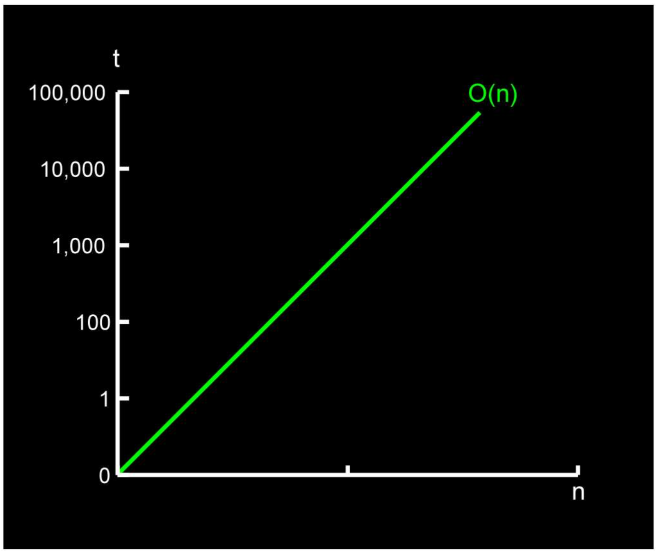

Figura 2: Gráfico de complexidade de algoritmo. Tempo de execução (eixo y) x numero de iteraçoes n (eixo x).

Complexidade Quadrática: $O(n^2)$

As operações a seguir seguem *complexidade quadrática*:

- **Realizando pesquisa linear em uma matriz.**
- **Ordenação do tipo *Quicksort*.**
- **Tipo de inserção.**

Exemplo de trecho de código:

```
void imprimeTodasCombinacoesElementos(int vetor[], int tamanhoVetor) {
    for (int i = 0; i < tamanhoVetor; i++) {
        for (int j = 0; j <tamanhoVetor; j++) {
            printf("Combinacao elemento[i] elemento[j] = %d%d\n", i, j, vetor[i],
vetor[j]);
        }
    }
}
```

No exemplo, aninhou-se dois laços (*loops*). O primeiro, que será executado i vezes, e, cada vez que o primeiro é executado, o segundo laço (*loop*) também é executado, j vezes. Portanto, ao declarar-se laços (*loop*) aninhados, a máquina haverá de executar um total de n * n vezes operações. Logo, conclui-se que a complexidade é $O(n * n)$, equivalente a dizer $O(n^2)$.

É recomendável que algoritmos que executem em tempo quadrático sejam evitados, uma vez que o tempo de execução aumenta quadráticamente. Por exemplo: para um tamanho de entrada de n=100.000 elementos, pode-se desprender 10 segundos para ser concluir-se a execução de um programa. Para um tamanho de entrada de n=1.000.000 elementos, ~16 min são desprendidos para concluir-se a execução, e, para um tamanho de entrada de n=10.000.000 elementos, poderia levar-se ~1,1 dias para concluir-se a execução do programa.

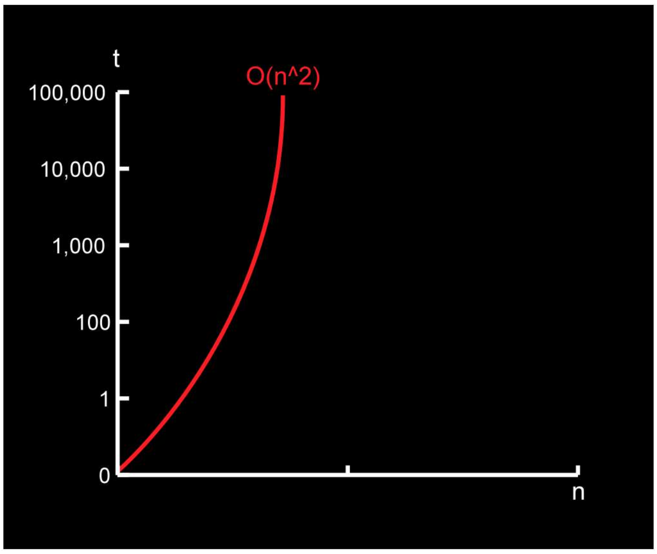

Figura 3:Gráfico de complexidade de algoritmo. Tempo de execução (eixo y) x numero de iteraçoes n (eixo x).

Complexidade Cúbica: O(n³)

As operações a seguir seguem *complexidade quadrática*:

- **Multiplicação de matrizes.**
- **Soluções de equações polinomiais de 3a ordem.**

Exemplo de trecho de código:

Obs: Assumindo-se que o número de colunas da primeira matriz (matriz1) é igual ao numero de linhas da segunda matriz (matriz2) e chamaremos essa variável de **dimensãoResult**.

```
void multiplicaMatrizes(int matriz1[][], intmatriz2[][], int dimensãoResult) {

    int x, i, j;
    int matrizResultado[dimensãoResult][dimensãoResult];
    for (i = 0; i < dimensãoResult; i++) {
        for (j = 0; j < dimensãoResult; j++) {
            matrizResultado[i][j] = 0;
            for (x = 0; x < dimensãoResult; x++) {
                matrizResultado[i][j] += matriz1[i][x] * matriz2[x][j];
```

```
            }
        }
    }
}
```

No exemplo, aninhou-se três laços (*loops*). O primeiro, que será executado i vezes, e, cada vez que o primeiro é executado, o segundo laço (*loop*) também é executado, j vezes, e, cada vez que o segundoo é executado, o terceiro laço (*loop*) também é executado, x vezes. Portanto, ao declarar-se laços (*loop*) aninhados, a máquina haverá de executar um total de (n * n * n) vezes operações. Logo, conclui-se que a complexidade é O(n * n * n), equivalente a dizer $O(n^3)$.

Complexidade Logarítmica: O(Log n)

Alguns exemplos comuns de *complexidade logarítmica* são:

- **Busca binária.**
- **Inserir ou excluir um elemento em um *heap*.**

Exemplo de trecho de código:

A *complexidade logarítmico* cresce mais devagar à medida que n cresce. Uma maneira fácil de verificar se um laço (*loop*) é log n é observar se a variável de contagem (neste caso: i) dobra, ao invés de incrementar em 1. No exemplo a seguir, int i não tem incremento 1 (i ++). Contrariamente à isso, esta variável dobra a cada execução do laço (*loop*), em tempo logarítmico. Portanto, tem complexidade log (n):

```
void testeComplLogn() {
    for(int i = 1; i < n; i *= 2) {
        printf("i = %d", i);
    }
}
```

Os algoritmos de complexidade temporal logarítmicos têm excelente desempenho em grandes conjuntos de dados:

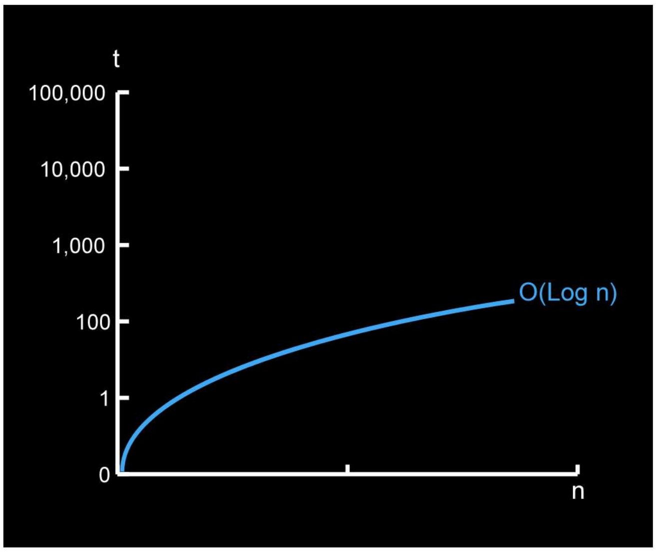

Figura 4: Gráfico de complexidade de algoritmo. Tempo de execução (eixo y) x numero de iteraçoes n (eixo x).

Complexidade O(n*Log n)

Alguns bons exemplos de algoritmos de complexidade O(n log n) são:

- **Heapsort**
- **Merge sort**
- **Quick sort**

Os algoritmos lineares são capazes de obter um bom desempenho com conjuntos de dados muito grandes.

Exemplo de trecho de código:

```
void merge_sort(int A[], int p, int r) {
    if (p < r) {
        q = (p + r) / 2;
        merge_sort(A, p, q);
        merge_sort(A, q+1, r);
        merge(A, p, q, r);
    }
}
```

Há a Invocação inicial da função: merge_sort(A,1, n), em que o vetor A contém n elementos.

O tamanho do input é uma potência de 2 e, em cada divisão, as subssequências tem tamanho exatamente = n/2.

Seja T(n) o tempo de execução sobre um input de tamanho n. Se n = 1, esse tempo é constante, que escrevemos $T(n) = \Theta(1)$. Se isso não ocorrer:

- Divisão: o cálculo da posição do meio do vector é feita em tempo constante: $D(n) = \Theta(1)$.

- Conquista: são resolvidos dois problemas, cada um de tamanho n/2, e, o tempo total para a execução é 2T(n/2).

- Combinação: a função *merge* executa em tempo linear: $C(n) = \Theta(n)$. Na função auxiliar a seguir:

```
void merge(int A[], int p, int q, int r) {
    int L[MAX], R[MAX];
    int n1 = q - p + 1;
    int n2 = r - q;
    for (i = 1; i <= n1; i++) {
        L[i] = A[p + i - 1];
    }
    for (j = 1; j <= n2; j++) {
        R[j] = A[q + j];
    }
    L[n1 + 1] = MAXINT;
    R[n2 + 1] = MAXINT;
    i = 1;
    j = 1;
    for (k = p; k <= r; k++) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        }
        else {
            A[k] = R[j];
            j++;
        }
    }
```

Então: $T(n) = \Theta(1)$ se $n = 1$ & $\Theta(1) + 2T(n/2) + \Theta(n)$ se $n > 1$.

Podemos reescrever para: $T(n) = \Theta(1)$ se $n \leq k$ & $D(n) + aT(n/b) + C(n)$ se $n > k$.

Cada etapa da Divisão gera sub-problemas de uma fração 1/b do original (pode ser b != a).

Logo: $T(n) = c$ se $n = 1$ & $2T(n/2) + cn$ se $n > 1$. Em que c é o maior entre os tempos necessários para resolver os problemas de dimensão 1 e o tempo de combinação por elemento dos vectores.

Então o custo total é $(\log n + 1)cn = cn \log n + cn$.

Portanto, o algoritmo "*merge sort*" executa (em qualquer caso) em $T(n) = \Theta(n \log n)$.

No gráfico a seguir, exemplifica-se o tempo desprendido na execução de tal algoritmo.

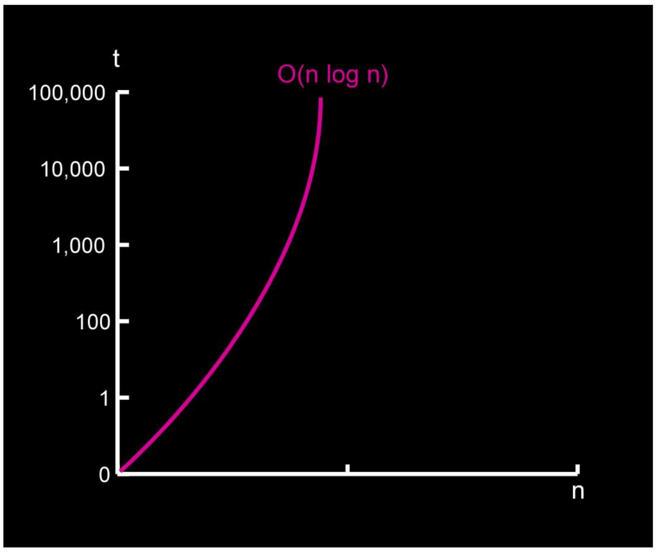

Figura 5: Gráfico de complexidade de algoritmo. Tempo de execução (eixo y) x numero de iteraçoes n (eixo x).