



Complexidade de Algoritmos

Prof. Sama Rouhani



Algoritmos

No mundo da ciência da computação, algoritmos são processos computacionais bem definidos que podem receber uma entrada, processá-la e produzir um ou mais valores como saída.

Algoritmos esses usados para resolver problemas.



Alguns exemplos de problemas que envolvem algoritmos comuns são:

- Calcular a rota mais curta entre duas casas
- Contar o número de amigos em comum em uma rede social
- Organizar de maneira eficiente tarefas de acordo com sua prioridade, prazo e duração
- Organizar a lista de clientes/fornecedores de acordo com uma ordem, ou prioridade, ou importância.
- Buscar uma mensagem no histórico de conversas



Mas o que é problema?

De forma geral, podemos pensar em algoritmos como uma ferramenta para resolver um problema bem definido.

Um problema se baseia em um conjunto de dados que se deseja processar, cada dado com sua especificidade (int, float, struct, set, union, etc), e o resultado é o que se deseja alcançar.



Tipos de algoritmos

Algumas das principais categorias/técnicas são:

- Greedy Algorithms (algoritmos gulosos)
- Dynamic Programming (programação dinâmica)
- Divide and conquer (dividir para conquistar)
- Backtracking (voltar para encontrar o melhor caminho)
- Search and Sorting (busca e ordenação)

Tipos de algoritmos

Algoritmos gulosos:

A estratégia gulosa tem por abordagem encontrar a melhor resposta para cada passo, sem se importar em resolver esse passo novamente ou com os passos seguintes, esperando como consequência um resultado global ótimo. Acabamos por ter algoritmos mais simples e intuitivos em grande parte dos casos, mas não necessariamente apresentando a melhor resposta.





Tipos de algoritmos

Programação dinâmica:

São algoritmos relacionados a problemas de **otimização**.

A solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada em uma **tabela**, de forma a evitar o recálculo.

Esse método, pode ser definido, vagamente, como recursão com o apoio de uma tabela: cada instância do problema é resolvida a partir da solução de instâncias menores, ou melhor, de subinstâncias da instância original.

A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias.

O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.



Tipos de algoritmos

Algoritmos dividir para conquistar:

São algoritmos que dividem o problema em problemas menores para poder chegar na solução unitária e então retornar passo a passo fazendo os cálculos necessários:

- Busca binária
- Merge sort
- Quicksort
- Mediana generalizada

Tipos de algoritmos

Backtracking

Backtracking é um **refinamento** do algoritmo de busca por força bruta (exaustiva), no qual boa parte das soluções podem ser eliminadas sem serem explicitamente examinadas .

Aplicações:

- Problema da mochila
- Problema das 8 rainhas
- Branch-and-Bound é uma técnica de exploração mais sofisticada, que procura explorar opções (branch), mas colocando um limite quantitativo (bound), com o objetivo de evitar buscas em espaços menos promissores
- Problema do caixeiro viajante: consiste em minimizar o custo de um caixeiro viajante que deseja percorrer n cidades, visitando cada cidade apenas uma vez, e retornar para casa.



Tipos de algoritmos

Search and Sorting

Temos diversos algoritmos de busca e ordenação

- **Busca:** sequential, binária, busca em largura e profundidade, etc
- **Ordenação:** bubble sort, quicksort, mergesort, etc




Cada problema tem a sua complexidade

Como analisar a complexidade de cada problema?

- Depende da linguagem a ser utilizada
- Depende do tipo de entrada (lista, pilha, inteiro, float)
- Depende da quantidade de entrada (10, 100 ou 1000000)
- Depende do poder de processamento da sua máquina
- Testando o código num cenário que por sorte ele roda rápido, mas em outro cenário ele pode rodar mais devagar.

O melhor algoritmo para resolver um problema é aquele que possui a **menor** complexidade de **tempo** e **espaço**. Em outras palavras, é o algoritmo que, conforme a entrada cresce tendendo ao infinito, é aquele que apresenta a menor variação de tempo e memória utilizada para terminar.



Cada problema tem a sua complexidade

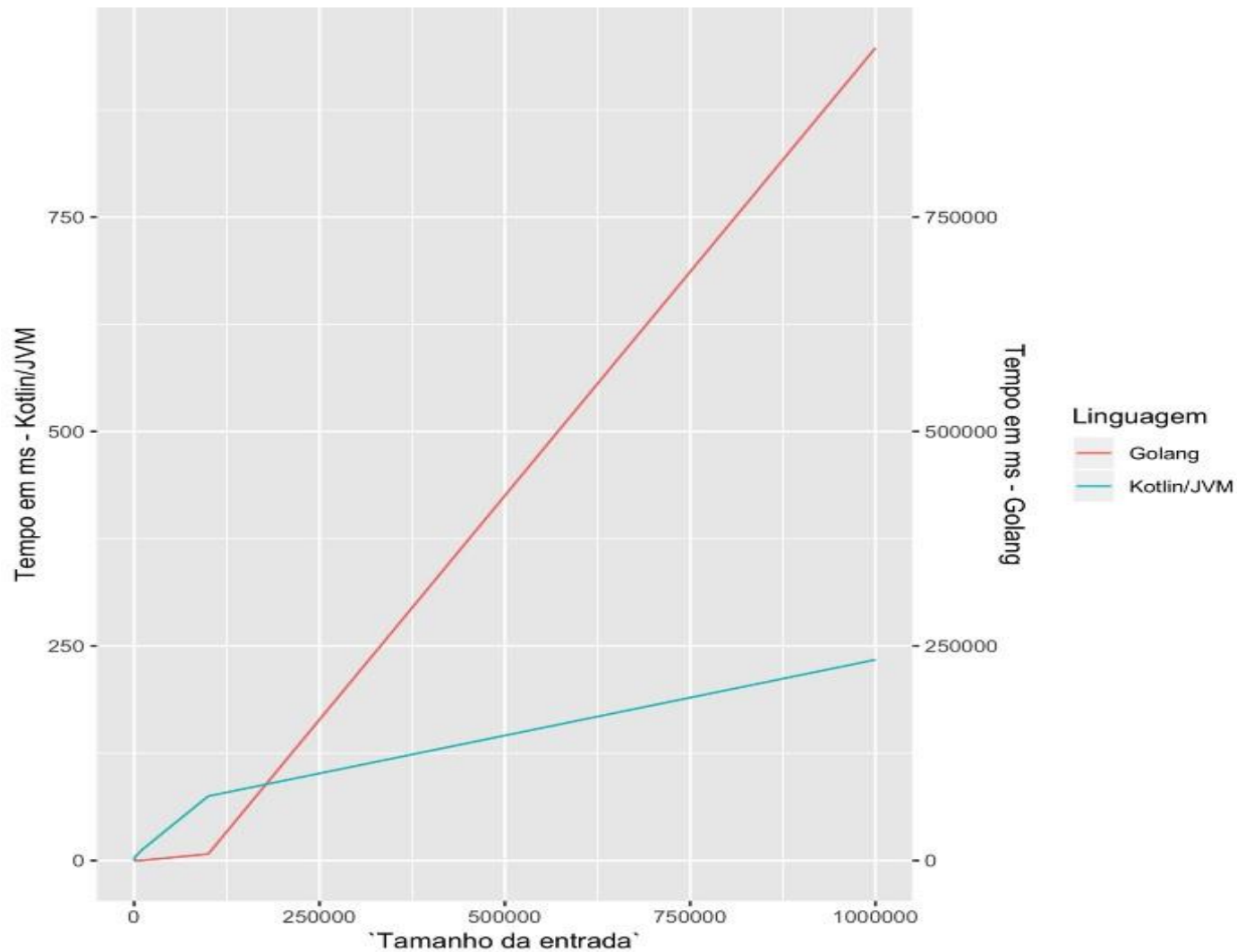
- **O custo final de um algoritmo pode estar relacionado a diversos fatores:**
 - Tempo de execução
 - Utilização de memória principal
 - Utilização de disco
 - Consumo de energia, de CPU, threads, etc
- **Medidas importantes em outros contextos:**
 - legibilidade do código
 - custo de implementação/manutenção
 - portabilidade
 - extensibilidade
 - documentação de código

Cada problema tem a sua complexidade

Uma boa idéia é estimar a eficiência de um algoritmo em função do tamanho do problema.

- Em geral, assume-se que “ N ” é o tamanho do problema, ou número de elementos que serão processados
- E calcula-se o número de operações que serão realizadas sobre os N elementos
- Deve-se preocupar com a eficiência de algoritmos quando o tamanho de N for grande.
- *A eficiência de um algoritmo descreve a eficiência relativa dele quando N se torna grande.*

Rodando os mesmos algoritmos com entradas variando de 0 a 1 milhão e extraíndo o tempo de execução podemos montar o seguinte gráfico:





Análise da Complexidade de Algoritmo

Um algoritmo pode ser melhor que outro quando processa poucos dados, porém pode ser muito pior conforme o dado cresce.

A **Análise de complexidade** nos permite medir o quão rápido um programa executa suas tarefas. Exemplos de tarefas são: Operações de adição e multiplicação; comparações; busca de elementos em um conjunto de dados; determinar o caminho mais curto entre diferentes pontos; ou até verificar a presença de uma expressão regular em uma string.



Análise da Complexidade de Algoritmo

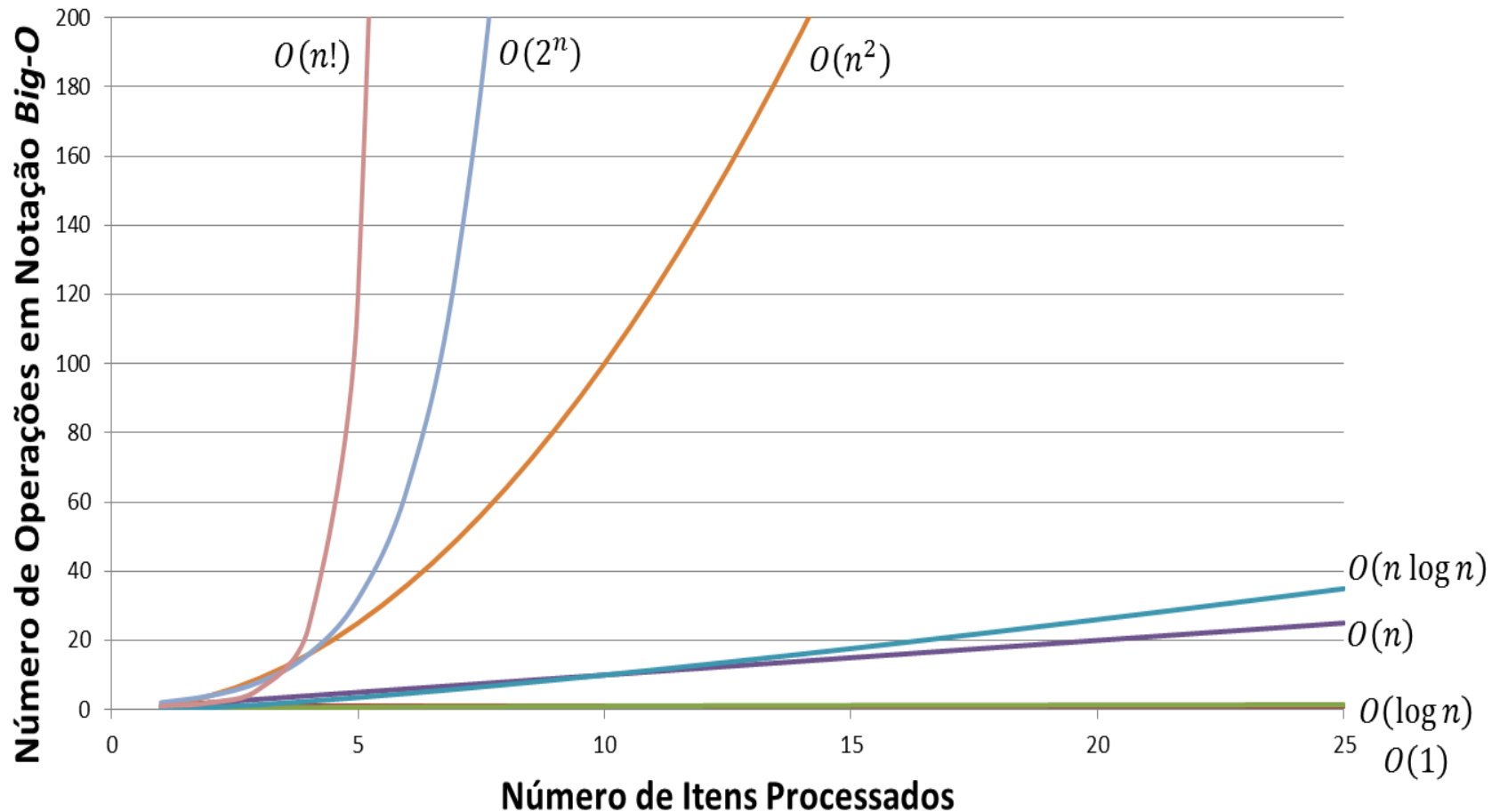
Em complexidade de algoritmos, a gente tenta analisar, de alguma forma, a quantidade de passos ou **iterações** que o nosso código leva para executar do início ao seu fim, considerando o pior caso possível, ou seja, o caso onde ele levaria o maior tempo e espaço possíveis.

A notação mais utilizada para descrevermos a complexidade de tempo nos nossos códigos é o **Big O**.

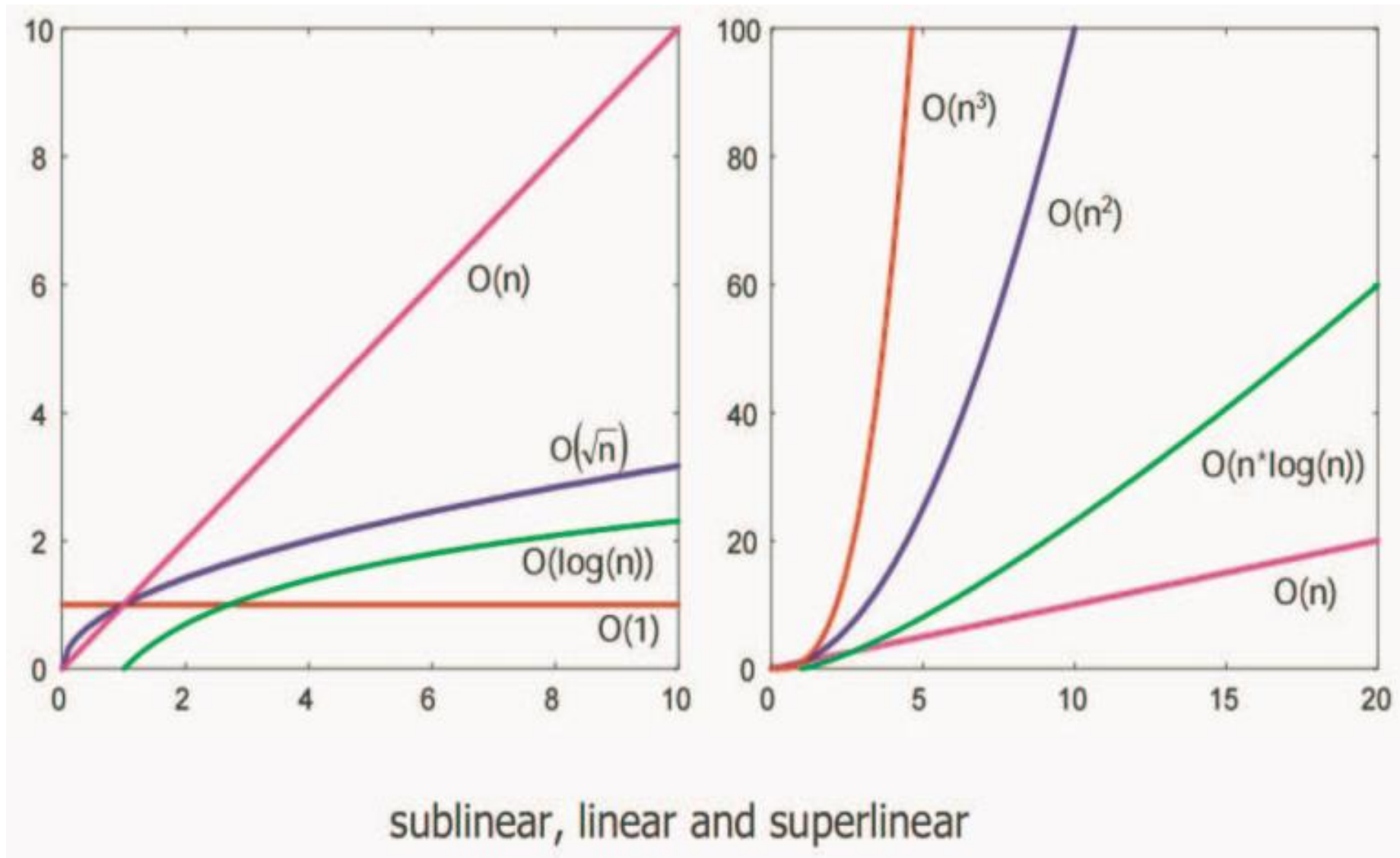
O **Big O** mostra o comportamento do nosso algoritmo em termos de crescimento de tempo de execução baseado nos seus valores de entrada.

Análise da Complexidade de Algoritmo

Ilustração das Complexidades Mais Comuns - Notação Big-O



Análise da Complexidade de Algoritmo





Análise da Complexidade de Algoritmo

$O(1)$ -> complexidade constante (independente do valor de entrada)

$O(\log N)$ -> quando tem algum loop de PA ou PG (fração)

$O(N)$ -> loop simples de N elementos

$O(N \log N)$ -> loop simples e loop de PA ou PG dentro do loop simples

$O(N^2)$ -> dois loops simples aninhados

$O(N^3)$ -> três loops simples aninhados

$O(2^n)$ -> exponencial

$O(N!)$ -> fatorial

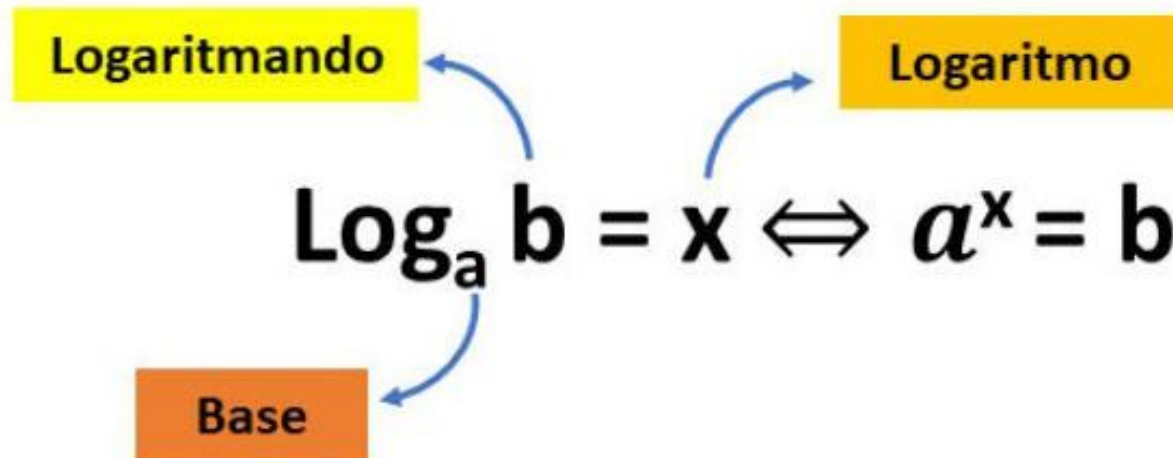
Análise da Complexidade de Algoritmo

Funções e taxas de crescimento

- Tempo constante: $O(1)$ (raro)
- Tempo sublinear ($\log(n)$): muito rápido (ótimo)
- Tempo linear: ($O(n)$): muito rápido (ótimo)
- Tempo $n \log n$: para algoritmos de divisão e conquista.
- Tempo polinomial n^k : Freqüentemente de baixa ordem ($k \leq 10$), considerado eficiente.
- Tempo exponencial: 2^n , $n!$, n^n considerados intratáveis

Análise da Complexidade de Algoritmo

Definição de logaritmo



Lê-se logaritmo de b na base a, sendo $a > 0$ e $a \neq 1$ e $b > 0$.

Quando a base de um logaritmo for omitida, significa que seu valor é igual a 10. Este tipo de logaritmo é chamado de logaritmo decimal.



Análise da Complexidade de Algoritmo

Portanto, 3 passos para calcular a complexidade:

- 1 - Levar em consideração apenas as repetições do código**
- 2 - Verificar a complexidade das funções ou métodos próprios da linguagem (se for utilizado)**
- 3 - Ignorar as constantes e utilizar o termo de maior grau**