FaculdadeBarretos
www.faculdadebarretos.com.br

Algoritmos e Estruturas de Dados

(Aula 8 - Introdução a análise de complexidade de algoritmos)

Prof. Me. Diogo Tavares da Silva
contato: *diogotavares@unibarretos.com.br*

FaculdadeBarretos
www.faculdadebarretos.com.br

Definição

"Análise da complexidade de algoritmos é o estudo que avalia o desempenho de um algoritmo em relação a tempo de processamento e recursos necessários para o processamento"

Questões - Contextualização

- Por que o meu programa está tão lento?
- Quanta memória meu programa vai utilizar?
- E se eu precisar aumentar o número de dados, este algoritmo será útil?
- Vou ter que comprar um computador melhor para ter uma resposta mais rapidamente?

Questões - Contextualização

Através da análise de algoritmos é possível:

- Determinar corretamente o uso de um algoritmo
- Comparar algoritmos
- Determinar o uso de processamento e memória
- Prever o crescimento dos recursos necessários às operações
- Relacionar o uso de recursos com as entradas
- Balancear o uso entre os recursos (espaço e tempo)

Como avaliar desempenho?

- **Análise de complexidade temporal**
  - A avaliação de desempenho de um algoritmo geralmente é feita de duas formas:
    - Empiricamente (Modelo Experimental)
      - Utilizando modelo experimental
    - Analiticamente (Modelo Matemático)
      - Através de análise teórica

Como avaliar desempenho?

- **Avaliação empírica**
  - Algoritmo é executado exaustivamente, com entradas diferentes e tamanhos diferentes
    - Calcula-se o tempo de processamento
      - Problemas:
        - Depende muito do hardware
        - Depende da linguagem e compilador escolhido

Como avaliar desempenho?

- **Avaliação por análise matemática**
  - Feita através de um modelo teórico
  - não existe a necessidade de que o algoritmo seja executado.
  - São considerados:
    - O tamanho da entrada (dados fornecidos)
    - O número de instruções realizadas pelo algoritmo

Análise de desempenho

- A eficiência de um algoritmo é calculada em função do tamanho do problema
  - ou seja, através do número de elementos que serão processados.
- **“n”** é o “tamanho do problema”, ou número de elementos que serão processados.
  - Calcula-se o número de operações que serão realizadas sobre os n elementos

Análise de desempenho

- deste modo…
- **Modelo Matemático**
  - Para uma entrada de tamanho n o tempo de execução é um somatório:
    - *Custo*Frequência de cada operação*
  - Caracteriza o tempo de execução como uma função em relação ao tamanho/tipo da entrada

Análise de complexidade por modelo matemático

- “Operações Básicas”
  - que operações considerar?
    - Alocação de variável
    - Atribuição
    - Comparação
    - Operação lógica
    - Operação aritmética
    - Acesso ao vetor
    - Acesso à variável
    - etc

Análise de complexidade por modelo matemático

- Estimativa de tempo de execução:
  - Analisar e definir quais operações serão consideradas/importantes (atribuição, comparação, etc.)
  - Considerar como constante o tempo de execução de cada operação
  - Aproximar estes valores utilizando uma função

Análise de complexidade por modelo matemático

- Estimativa de tempo de execução:
  - Considerar três casos:
    - **Pior caso**
    - Caso médio
    - Melhor caso

Análise de complexidade por modelo matemático

- Estimativa de tempo de execução:
  - Considerar três casos:
    - **Pior caso**
    - Caso médio
    - Melhor caso

Ex:

- Quantas instruções são executadas nesse código?

```
1. for( int i = 0; i < n; i++)
2.     vetor[i] = i + 1;
```

Operação	Frequência
Alocação de variável	1
Atribuição	N+1
Comparação menor	N+1
Incremento	N
Soma	N
Acesso de vetor	N

Outro ex:

- Quantas instruções são executadas nesse código?

```
1. int count = 0;
2. for(int i = 0; i < n; i++)
3.     if( vetor[i] == 0 )
4.         count++;
```

Operação	Frequência
Alocação de variável	2
Atribuição	2
Comparação menor	N+1
Incremento	N até 2*N
Comparação igual	N
Acesso de vetor	N

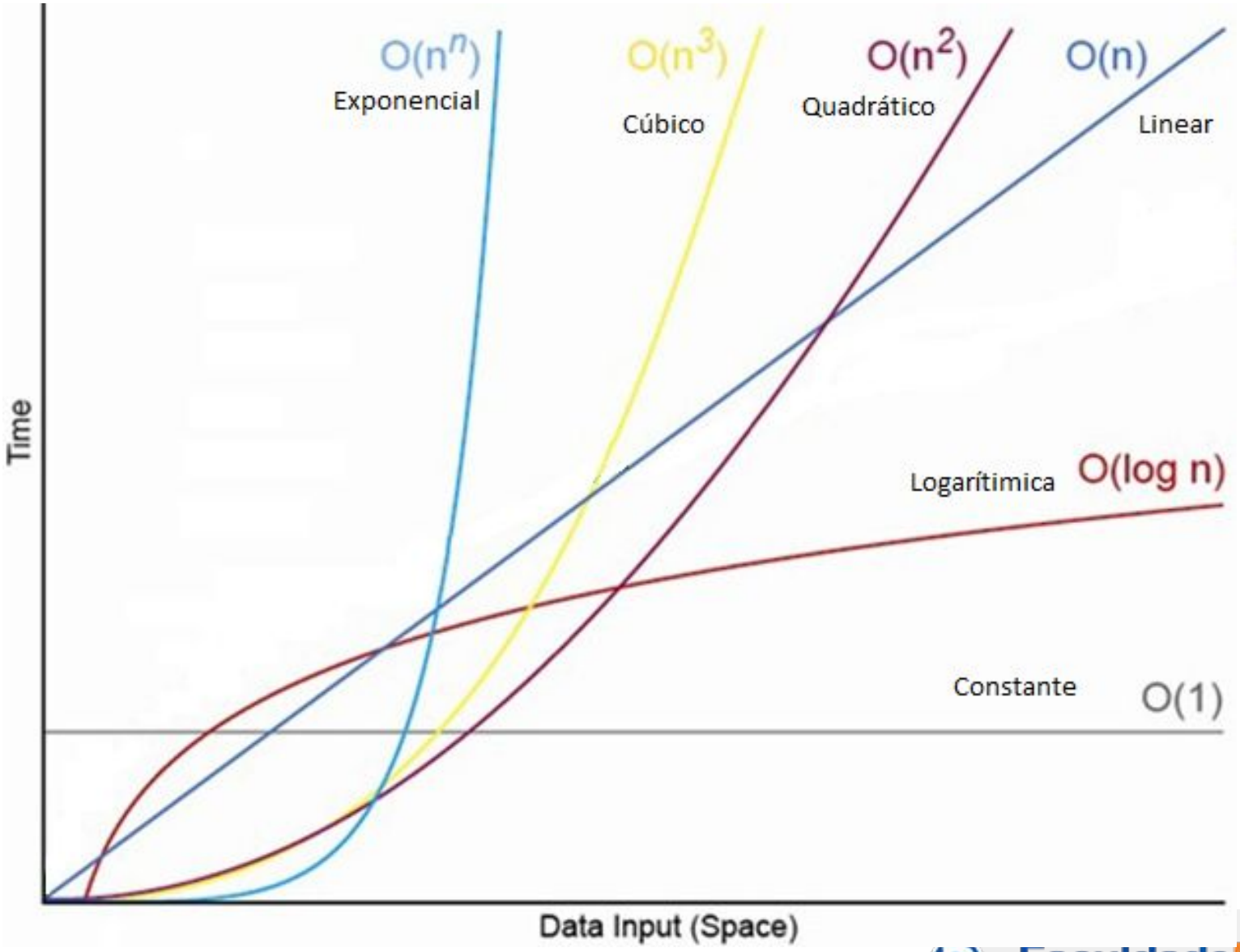
Notação til (~)

- Ao realizar a análise, ignorar os termos de menor ordem
  - a taxa de crescimento não é afetada por eles
    - justificativa:
    - **Quando o N é grande:**
      - os termos de menor ordem não mudam muito o resultado
    - **Quando o N é pequeno:**
      - A entrada é pequena, causando pouca diferença

Notação til (~)

- exemplo:
$1/6\ N^3 + 20\ N + 16$	$\sim 1/6\ N^3$
$3N^2 + 100\ N + 3$	$\sim 3\ N^2$
$1/6\ N^3 - ½\ N^2 + 1/3\ N$	$\sim 1/6\ N^3$

Ordem de crescimento



Ordem de crescimento

- Conjunto pequeno de funções:
  - Constante: ~1
  - Logarítmica: ~log(N)
  - Linear: ~N
  - Linear-Logarítmica: ~N*log(N)
  - Quadrática: ~N^2
  - Cúbica: ~N^3
  - Exponencial: ~2^N , n!, n^n

Tipos de análise

- **Melhor caso**
  - Quando os dados de entrada que levam ao menor número de operações que serão executadas
- **Pior caso**
  - Quando os dados de entrada levam ao maior número de operações. Considerando qualquer entrada, este será o maior tempo necessário de processamento
- **Caso médio**
  - Custo estimado médio. Muito difícil de prever

Análise e notação assintótica

- Consideram o comportamento das funções de desempenho temporal/espacial para quando o tamanho da entrada n é muito grande
  - ou seja, quando $n \rightarrow \infty$

Análise e notação assintótica

- Exemplo: Dois algoritmos resolvem o mesmo problema com diferentes números de operações

- Algoritmo 1: $f1(n) = 2n^2 + 5n$ operações
- Algoritmo 2: $f2(n) = 500n + 4000$ operações

quando $n \rightarrow \infty$

Alg 1 = $f1(n) = 2n^2$

Alg 2 = $f2(n) = 500n$

Análise e notação assintótica

- Exemplo: Dois algoritmos resolvem o mesmo problema com diferentes números de operações

- Algoritmo 1: $f1(n) = 2n^2 + 5n$ operações
- Algoritmo 2: $f2(n) = 500n + 4000$ operações

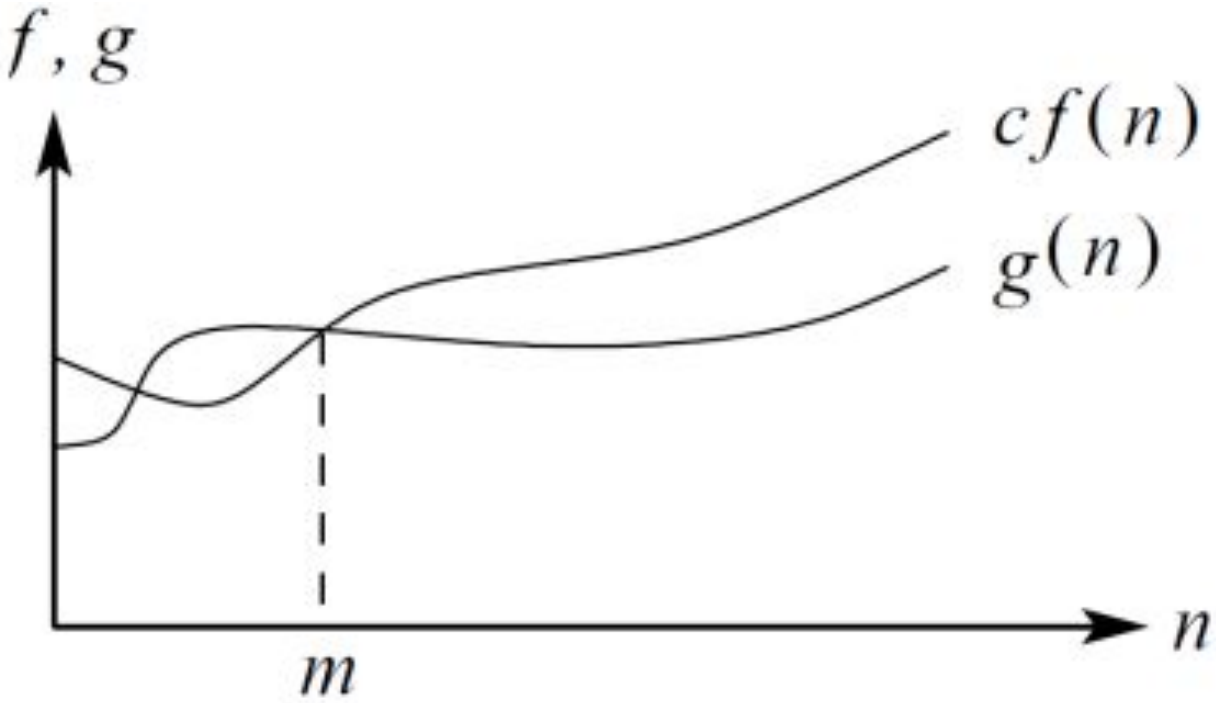
quando $n \rightarrow \infty$

Alg 1 = $f1(n) = 2n^2$

Alg 2 = $f2(n) = 500n$

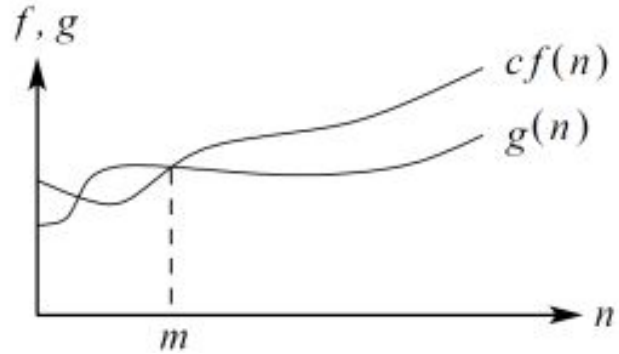
Análise e notação assintótica

Uma função f(*n*) domina assintoticamente outra função g(*n*) se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c|f(n)|$



Notação O (“Big-Oh”)

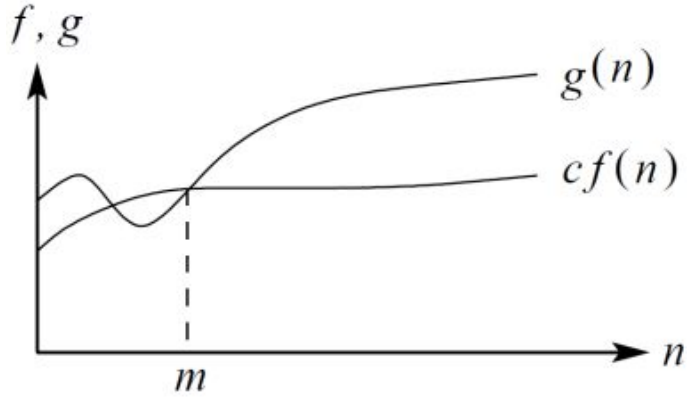
- Define g(*n*) = O(f(*n*)) se f(*n*) domina assintoticamente g(*n*)
- g(n) tem taxa de crescimento proporcional a f(n), ou seja, é de **ORDEM MÁXIMA** f(n).
- *f* expressa um LIMITE SUPERIOR para valores assintóticos de g, ou seja, O(f(n)) NUNCA excede c*f(n) (se n for suficiente grande)



NOTAÇÃO Ω (Big Omega)

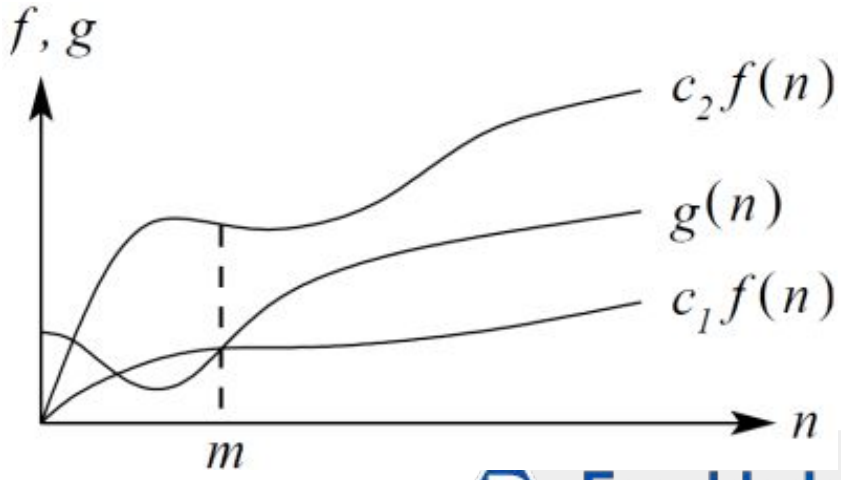
Uma função g(*n*) é Ω(f(*n*)) se g(*n*) domina assintoticamente f(*n*)

- **Notação O denota um limite superior**
- **Notação Ω denota um limite inferior**



NOTAÇÃO Θ (Theta)

- Uma função g(*n*) é Θ(f(*n*)) se g(*n*) e f(*n*) dominam assintoticamente uma à outra
- Duas funções crescem de forma similar, mantendo a diferença (limitada por cima e por baixo)



Notações assintóticas

- **O** é mais utilizada para avaliar o pior caso
- **Ω** é geralmente mais utilizada para avaliar o melhor caso
- **Θ** indica limite superior e inferior a função

Funções e Taxas de crescimento

- **Tempo constante**: O(1) (raro)
- **Tempo logarítmico (log(*n*))**: muito rápido (ótimo)
- **Tempo linear**: (O(*n*)): muito rápido (ótimo)
- **Tempo *n* log *n***: Comum em algoritmos de divisão e conquista (eficaz)
- **Tempo polinomial n^k** : Frequentemente de baixa ordem ($k \leq 10$), considerado eficiente
- **Tempo exponencial**: 2^n , $n!$, n^n considerados intratáveis

Funções e Taxas de crescimento

F(n) = O(1)

- **Complexidade constante**
- Tempo de execução do algoritmo independe do tamanho da entrada
- Os passos do algoritmo são executados um número fixo de vezes

Ex: determinar se um número é ímpar, inserir na cabeça da lista, acessar um elemento do vetor

Funções e Taxas de crescimento

F(n) = O(log(n))

- **Complexidade logarítmica**
- Típico de algoritmos “dividir para conquistar”
- Tempo de execução pode ser considerado menor do que uma constante grande
- Quando *n* é um milhão, log(*n*) ≈ 20 A base do logaritmo tem impacto pequeno

Exemplo: busca binária

Funções e Taxas de crescimento

F(n) = O(n)

- **Complexidade linear**
- O algoritmo realiza um número fixo de operações sobre cada elemento da entrada
- Melhor situação para um algoritmo que processa *n* elementos de entrada e produz *n* elementos de saída

Exemplo: busca sequencial, percurso em vetor, percurso em lista

Funções e Taxas de crescimento

F(n) = O(n log(n))

- **Complexidade linear logarítmica**
- Típico de algoritmos que dividem um problema em subproblemas, resolve cada subproblema de forma independente, e depois combina os resultados

Exemplo: ordenação (eficiente) (*algoritmos quicksort, mergesort*)

Funções e Taxas de crescimento

$F(n) = O(n^2)$

- **Complexidade quadrática**
- Típico de algoritmos que operam sobre pares dos elementos de entrada
- Comumente em um loop aninhado
- Útil para resolver problemas pequenos

Exemplos: ordenação (ineficiente)(*bubblesort, insertion sort, selection sort*); percurso em matriz

Funções e Taxas de crescimento

$F(n) = O(c^n)$

- **Complexidade exponencial**
- Típicos de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema
  - Algumas abordagens recursivas
- Problemáticos e pouco úteis do ponto de vista prático

Quando *n* é 20, $O(2^n)$ é ≈ um milhão