# Análise de Complexidade

Rafael Castro

Curso Preparatório para ED

# Função de Complexidade

- Descreve o “custo” de executar um algoritmo dado o tamanho da entrada (o tal do “n”)
- Geralmente remete ao “Pior Caso”, mas também pode ser referente ao “Caso Médio”
  - “Melhor Caso” é válido, mas raramente útil

- Todo algoritmo possui um “custo”
  - Tempo de processamento
  - Alocação de memória
  - Operações lentas (ler arquivo)
  - Tempo online no AWS

- Abstrai muito dos detalhes de implementação

# O que se presume?

- Analisando o tempo de execução
  - Todas operações tem o mesmo custo
  - Não há interferência do Sistema Operacional
  - Não acontecem erros

- Analisando a memória
  - O sistema nunca fica sem espaço
  - Não inclui memória do sistema

# Exemplo: Busca Maior elemento

```
A: Array // tamanho = n

func find_max(A):
    max = A[0] // c1
    for elemento in A: // roda n vezes
        if elemento > max: // c2
            max = elemento // c3
    return max // c4
```

F(n) = c1 + c4 + n*(c2 + c3) = **c5 + c6*n ≈ n**

# Detalhe Importante: Tamanho

- Complexidade é sempre em termos relativos ao **tamanho** da entrada

- Mas não pode se pode presumir o formato dessa entrada
  - Caso seja feito, sua complexidade vai se aplicar a uma versão “restrita” do problema

- O tamanho pode ser separado dependendo da entrada
  - Exemplo: Algoritmo recebe dois arrays diferentes como entrada

# Exemplo: Busca Por Elemento (Sequencial)

```
A: Array, buscado: Int

func find_normal(A, buscado):
    size = A.size()
    for indice in 0..size: // máximo n vezes
        elemento = A[indice]    // cada operação
        if elemento == buscado:// executa apenas 1 vez
            return indice
```

Complexidade (pior caso): **F(n) = n**

# Mas e o caso médio?

```
A: Array, buscado: Int

func find_normal(A, buscado):
    size = A.size()
    for indice in 0..size:
        elemento = A[indice]
        if elemento == buscado:
            return indice
```

- Supondo que o elemento sempre está no array
- E que ele tem probabilidade igual de estar em qualquer posição
- A complexidadedo caso médio é **F(n) = (n+1)/2**

# Análise Assintótica

- Análise de funções para “valores arbitrariamente grandes”
  - Semelhante ao estudo de limites em cálculo

- Separa funções em conjuntos chamados “Ordens de Complexidade”

- Pode ser simplificada focando nos fatores “dominantes”
- Facilita comparação de algoritmos

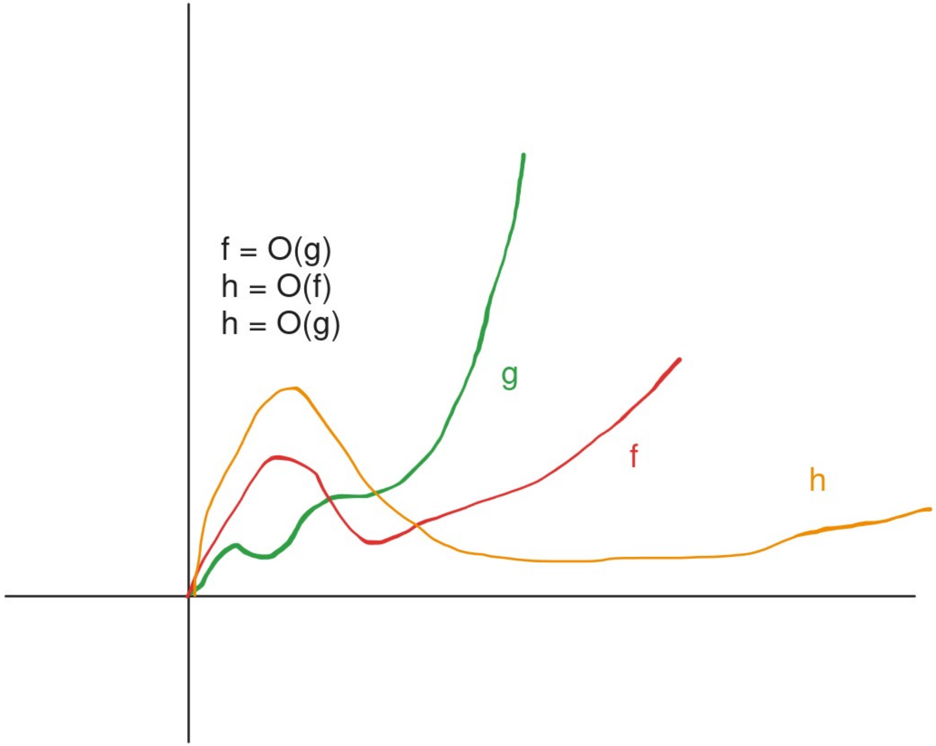
# Notação de Ordens de Complexidade

- Define uma classe de funções com base em uma função “limite”

- Existem 3 tipos principais:
  - O (“ó grande”) e Ω(“Omega grande”) são opostos
  - Θ (“téta grande”) é a união dos dois

- Para dizer que uma função f pertence a “ó grande” de outra g:
  - $f(n) = O(g(n))$

# Exemplo “Intuitivo”




- Nesse exemplo, a partir de um certo ponto:

- F nunca vai superar G
- H nunca vai superar F
- H nunca vai superar G

# Definição Formal

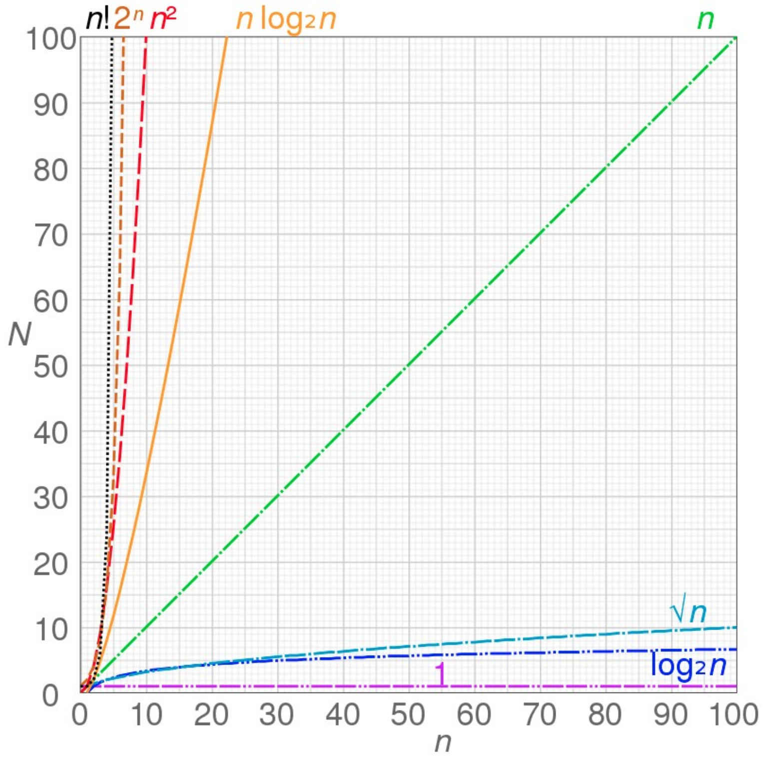
- Útil pra provar complexidades na prova!
- Dadas duas funções f(n) e g(n), podemos dizer que f(n) = O(g(n))
- Se
  - A partir de algum n' arbitrário, exista uma constante C tal que:
    - $f(n) < C*g(n)$
    - Se $n>n'$
- O mesmo vale pra Ω, trocando apenas a comparação para ">"

- f(n) = Θ(g(n)) se e somente se f for O(g(n)) **e** Ω(g(n))

# Divisão em Classes

- Ordens de complexidade dividem as funções em classes

- Existe uma “hierarquia de funções” mais comuns
  - As “piores” são as exponenciais
  - As “melhores” são as logaritmicas



# Na Prática

- A grande maioria dos algoritmos usáveis são “Polinomiais”
  - Da ordem O(n^x) onde x é um número positivo
  - O termo polinomial de maior expoente “domina” os menores
  - $F(n) = n + 100000*n^2 + 0.00001*n^3 = \boldsymbol{\Theta(n^3)}$

- Funções logaritmicas são mais “lentas” que qualquer polinômio

- Exponenciais são raras em algoritmos usáveis
  - Rapidamente se tornam “intratáveis

# Na prática: Busca Maior

```
A: Array // tamanho = n

func find_max(A):
    max = A[0] // c1
    for elemento in A: // roda n vezes
        if elemento > max: // c2
            max = elemento // c3
    return max // c4
```

$F(n) = c1 + c4 + n*(c2 + c3) = \mathbf{c5 + c6*n \approx n = O(n)}$

# Na Prática: Caso Médio da Busca Sequencial

```
A: Array, buscado: Int

func find_normal(A, buscado):
    size = A.size()
    for indice in 0..size:
        elemento = A[indice]
        if elemento == buscado:
            return indice
```

- A complexidade é

  $F(n) = (n+1)/2$

- $(n+1)/2 = \Theta(n)$
- Portanto a ordem de complexidade do caso médio é $\Theta(n)$

# Um exemplo mais complexo: InsertionSort

```
A: Array
func sort(A):
    for i in 0..(A.size-1): // executa de 0 até n-1
        // função find_max vista anteriormente: O(n)
        max = find_max(A[i:]) //maior elemento a partir de i
        troca(A[max], A[i]) // coloca maior elemento em i
```

Executa n vezes uma função com complexidade $\Theta(n)$

Complexidade: **$f(n) = \Theta(n^2)$**

# Foi isso

# Caso dê tempo: Teorema Mestre

- Teorema usado pra funções recursivas
  - Particularmente as do tipo “dividir para conquistar”

- Simplifica bastante o processo de análise de funções assim

- Possui 3 casos onde se aplica

# Exemplo: Busca Binária

```
A: Array, buscado: Int // A é ordenado
func find_binario(A, buscado, comeco, fim):
    meio = (comeco+fim)/2
    if buscado == A[meio]:
        return meio
    elif buscado < A[meio]:
        return comeco + find_binario(A, buscado, meio, fim)
    else:
        return meio + find_binario(A, buscado, meio, fim)
```

# Os casos do teorema mestre

- Seja **a** o número de vezes que uma função se chama
- Seja **b** o fator pelo qual a entrada é dividida
- Seja f(n) a complexidade não-recursiva de uma chamada

- O teorema mestre avalia a expressão de complexidade da forma

$$T(n) = aT(n/b) + f(n)$$

$$T(0) = 1$$

# Os 3 casos

- Sempre compara a função não-recursiva f(n) com o custo recursivo
  - Custo recursivo é dado na forma $n^c$, sendo $c=\log_b(a)$
- Uma versão simplificada:
  - Se $f(n) < n^c$, $T(n) = \Theta(n^c)$
  - Se $f(n) = n^c$, $T(n) = \Theta(f(n)*\log(n))$
  - Se $f(n) < n^c$, $T(n) = \Theta(f(n))$
- Mas existem detalhes importantes!