Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

Noções de Análise de Complexidade

Prof. Mateus Ferreira Satler

Índice

1. Introdução

- **Análise de um algoritmo particular:**
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - Características que devem ser investigadas:
    - Análise do número de vezes que cada parte do algoritmo deve ser executada,
    - Estudo da quantidade de memória necessária.

1. Introdução

- **Análise de uma classe de algoritmos:**
  - Qual é o algoritmo de menor custo possível para resolver um problema particular?
  - Toda uma família de algoritmos é investigada.
  - Procura-se identificar um que seja o melhor possível.
  - Coloca-se limites para a complexidade computacional dos algoritmos pertencentes à classe.

1. Introdução

- **Custo de um Algoritmo**
  - Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema.
  - Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.
  - Podem existir vários algoritmos para resolver o mesmo problema.
  - Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

1. Introdução

- **Medida do Custo pela Execução do Programa**
  - Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
    - Os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras.
    - Os resultados dependem do hardware.
  - Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
    - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza.
    - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

1. Introdução

- **Medida do Custo por meio de um Modelo Matemático**
  - Usa um modelo matemático baseado em um computador idealizado: **Random Access Machine (RAM)**
    - Para cada algoritmo, permite definir uma **função**, com base na entrada, para estimar o tempo gasto.
  - Deve ser especificado o conjunto de operações e seus custos de execuções.
    - É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
  - Ex.: Algoritmos de ordenação
    - Consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

1. Introdução

- **Função de Complexidade**
  - Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade *f***.
  - ***f(n)*** é a medida do tempo necessário para executar um algoritmo para um problema de tamanho ***n***.
  - Função de **complexidade de tempo**: *f(n)* mede o tempo necessário para executar um algoritmo em um problema de tamanho *n*.
  - Função de **complexidade de espaço**: *f(n)* mede a memória necessária para executar um algoritmo em um problema de tamanho *n*.

1. Introdução

- **Função de Complexidade**
  - Utilizaremos f para denotar uma **função de complexidade de tempo** daqui para a frente.
  - A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

1.1. Exemplo

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros A[n], onde n >= 1.

```
int max(int* A, int n){
  int i, temp;

  temp = A[0];
  for(i=1; i<n; i++)
    if(temp < A[i])
      temp = A[i];

  return temp; }
```

- Seja *f* uma função de complexidade tal que *f(n)* é o número de comparações entre os elementos de A, se **A** contiver **n** elementos.

- Qual a função *f(n)*? $f(n) = n-1$

1.1. Exemplo

- **Teorema:** Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, n >= 1, faz pelo menos n-1 comparações.

- **Prova:** Cada um dos n-1 elementos tem de ser investigado por meio de comparações, que é menor do que algum outro elemento.
  - Logo, n-1 comparações são necessárias

- O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então a função **max** do programa anterior é **ótima**.

1. Introdução

- A **medida do custo de execução** de um algoritmo depende principalmente do **tamanho da entrada dos dados**.
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - No caso da função `max` do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.
  - Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

2. Melhor Caso, Pior Caso e Caso Médio

- Imagine agora uma função que retorna "1" se um determinado elemento pertence a um vetor, e retorna "0" caso contrário.
  - Depende da **posição** desse elemento dentro do vetor.

- **Melhor caso:** o elemento está na primeira posição do vetor. Menor tempo de execução sobre todas as entradas de tamanho n.

- **Pior caso:** o elemento estar na última posição do vetor. Maior tempo de execução sobre todas as entradas de tamanho n.
  - Se *f* é uma função de complexidade baseada na análise de **pior caso**, o **custo** de aplicar o algoritmo nunca é maior do que ***f(n)***.

- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.

2. Melhor Caso, Pior Caso e Caso Médio

- Na análise do caso médio esperado, supõe-se uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n e o custo médio é obtido com base nessa distribuição.
- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis.
- Na prática isso nem sempre é verdade.

2.1. Exemplo de Cálculo de Caso Médio

- Considere o problema de encontrar um número de matrícula em um vetor de matrículas.
- Cada posição do vetor contém uma matrícula única.
- **O problema**: dada uma matrícula qualquer, localize a posição do vetor que contenha esta matrícula.
- O algoritmo de pesquisa mais simples é o que faz a **pesquisa sequencial**.

2.1. Exemplo de Cálculo de Caso Médio

- Seja ***f*** uma função de complexidade tal que ***f(n)*** é a quantidade de matrículas consultados no vetor (número de vezes que se compara a matrícula buscada com as matrículas armazenadas no vetor):
  - **Melhor caso:**
    - A matrícula procurada é a primeiro consultada
    - $f(n) = 1$
  - **Pior caso:**
    - A matrícula procurada é a última consultada ou não está presente no vetor
    - $f(n) = n$

2.1. Exemplo de Cálculo de Caso Médio

- **Caso Médio:**
  - No estudo do caso médio, vamos considerar que toda pesquisa recupera uma matrícula.
  - Se p_i for a probabilidade de que a i–ésima matrícula seja procurada, e considerando que para recuperar a i–ésima matrícula são necessárias i comparações, então:

$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$$

2.1. Exemplo de Cálculo de Caso Médio

- **Caso Médio:**
  - Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i
  - Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então:

$$p_i = \frac{1}{n}, 1 \leq i \leq n$$

  - Nesse caso: $f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$
  - A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

3. Análise Assintótica

- O parâmetro n fornece uma medida da dificuldade para se resolver o problema.
- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
  - A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- Logo, a análise de algoritmos é realizada para valores grandes de n.
- Estuda-se o comportamento das **funções de custo *f(n)***.

3. Análise Assintótica

- Precisamos de ferramentas matemáticas para **comparar funções.**

- Na análise de algoritmos usa-se a **Análise Assintótica.**
  - Estudo do comportamento de algoritmos para entradas arbitrariamente grandes ou a "descrição" da taxa de crescimento.

- Permite "simplificar" expressões como as mostradas anteriormente focando apenas nas **ordens de grandeza.**

3. Análise Assintótica

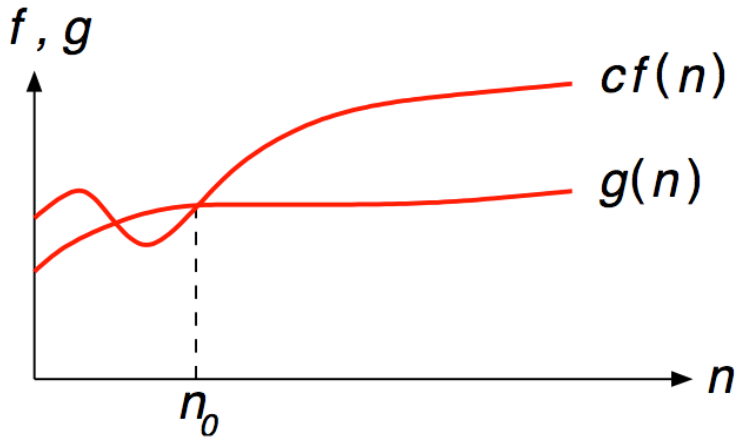
- Para entradas grandes o bastante, as constantes multiplicativas e os termos de mais baixa ordem de um tempo de execução podem ser ignorados.

- Considerando a função $f(n) = 3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

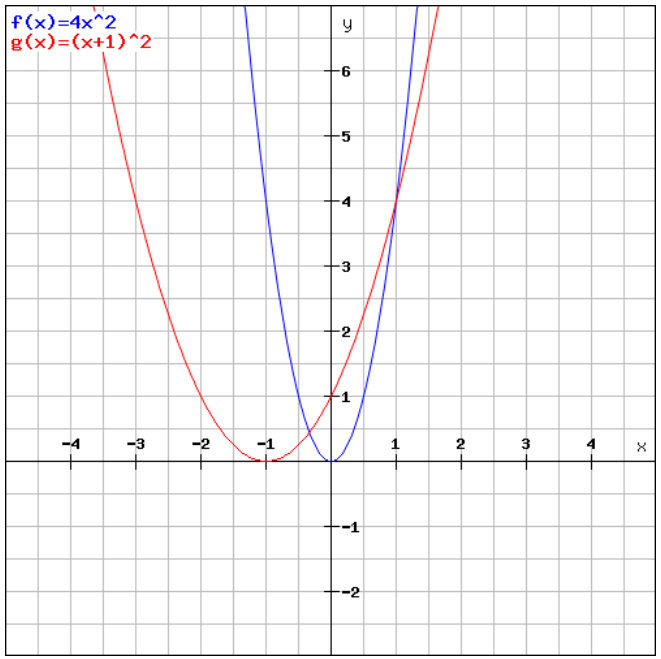
3. Análise Assintótica

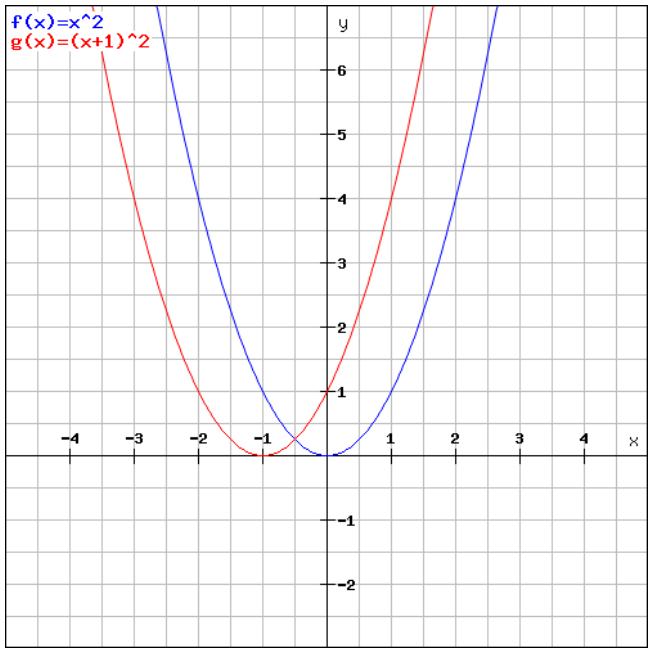
- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- **Definição**: Uma função *f(n)* **domina assintoticamente** outra função *g(n)* se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$ temos $|g(n)| \leq c \times |f(n)|$



3. Análise Assintótica

- Exemplo:
  - Sejam $g(n) = (n+1)^2$ e $f(n) = n^2$
  - As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, já que:
    - $|(n+1)^2| \leq 4\,|(n^2)|$ para $n \geq 1$ e
    - $|(n^2)| \leq |(n+1)^2|$ para $n \geq 0$





3. Análise Assintótica

- É interessante comparar algoritmos para valores grandes de n.
- O custo assintótico de uma função T(n) representa o limite do comportamento de custo quando n cresce.
- Em geral, o custo aumenta com o tamanho n do problema.
- **Observação**:
  - Para valores pequenos de n, mesmo um algoritmo ineficiente não custa muito para ser executado.

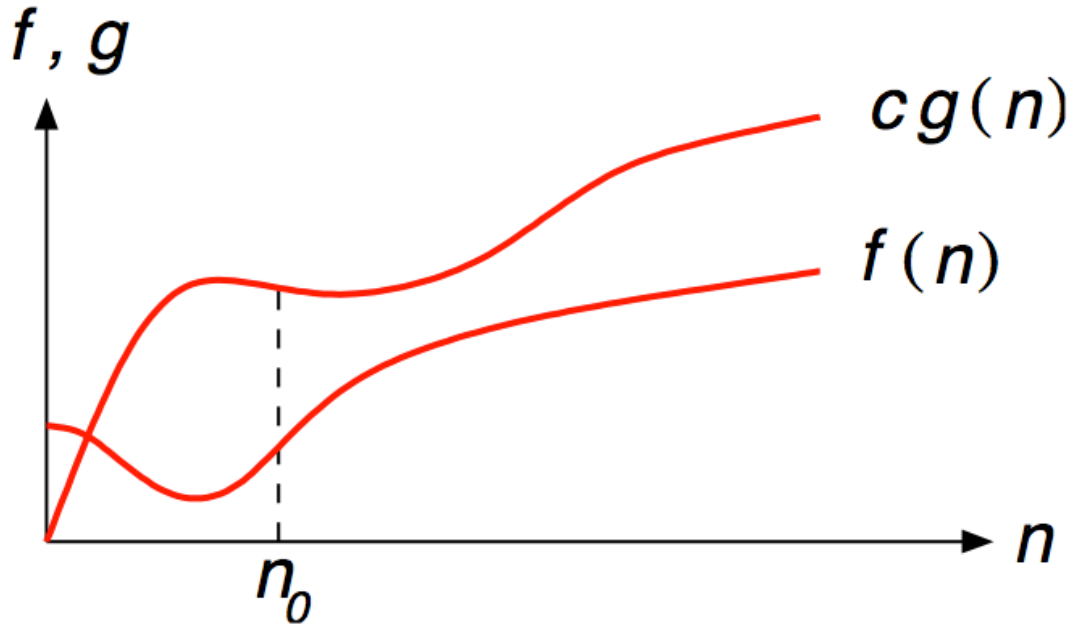
3. Análise Assintótica

- **Notação assintótica de funções**
  - Existem três notações principais na análise assintótica de funções:

1. **Notação O (“O” grande, big “O”)**
2. **Notação Ω (ômega)**
3. **Notação Θ (theta)**

4. Notação O

- **f(n) = O(g(n))**
  - Significa que ***c* × *g(n)*** é um limite superior de ***f(n)***



4. Notação O

- A notação **O** define um **limite superior** para a função, por um fator constante.
- Escreve-se $f(n) = O(g(n))$, se existirem constantes positivas **c** e $\mathbf{n_0}$ tais que para $\mathbf{n \geq n_0}$, o valor de $f(n)$ é menor ou igual a $c \times g(n)$.
  - Pode-se dizer que $g(n)$ é um **limite assintótico superior** (em inglês, asymptotically upper bound) para f(n)

$$f(n) = O(g(n)) \text{ , } \exists\ c > 0 \text{ e } n_0 \mid 0 \leq f(n) \leq c \times g(n), \forall n \geq n_0$$

- Escrevemos $f(n) = O(g(n))$ para expressar que ***g(n)* domina assintoticamente *f(n)***.
  - Lê-se f(n) é da ordem no máximo g(n).
- Observe que a notação **O** define um conjunto de funções:

$$O(g(n)) = \{ f: \aleph \rightarrow \Re^+ \mid \exists\ c > 0,\ n_0,\ 0 \leq f(n) \leq c \times g(n),\ \forall n \geq n_0 \}$$

4. Notação O

- Exemplos:

1. Seja $f(n) = (n + 1)^2$
   - Logo $f(n)$ é $O(n^2)$, quando $n_0 = 1$ e $c = 4$, já que $(n+1)^2 \leq 4n^2$ para $n \geq 1$

2. Seja $f(n) = n$ e $g(n) = n^2$. Mostre que $g(n)$ **não é** $O(n)$.
   - Sabemos que $f(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$.
   - Suponha que existam constantes c e n_0 tais que para todo $n \geq n_0$, $n^2 \leq c \times n$.
   - Assim, $\boldsymbol{c \geq n}$ para **qualquer** $n \geq n_0$.
   - No entanto, não existe uma constante $\boldsymbol{c}$ que possa ser maior ou igual a $\boldsymbol{n}$ **para todo** $\boldsymbol{n}$.

4. Notação O

- Quando a notação **O** é usada para expressar o tempo de execução de um algoritmo no pior caso, está se definindo também o **limite superior do tempo** de execução desse algoritmo para **todas as entradas**.

- Por exemplo, o algoritmo de ordenação por inserção é $O(n^2)$ no pior caso.
  - Este limite se aplica para **qualquer** entrada.

4. Notação O

- **Operações com a notação O**

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \quad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

4. Notação O

- **Operações com a notação O: Exemplos**
  - **Regra da soma *O(f(n)) + O(g(n))***
    - Suponha três trechos cujos tempos de execução sejam $O(n)$, $O(n^2)$ e $O(n \log n)$
    - O tempo de execução dos dois primeiros trechos é $O(max(n,n^2)) = O(n^2)$
    - O tempo de execução de todos os três trechos é então $O(max(n^2, n \log n))$ que é $O(n^2)$

4. Notação O

- **Regras Práticas:**
  - **Multiplicação por uma constante** não altera o comportamento:
    - $O(c \times f(n)) = O(f(n))$
    - $99 \times n^2 = O(n^2)$
  - Em um polinômio $a_x n^x + a_{x-1} n^{x-1} + \ldots + a_2 n^2 + a_1 n + a_0$ podemos nos focar na parcela com o **maior expoente**:
    - $3n^3 - 5n^2 + 100 = \mathbf{O(n^3)}$
    - $6n^4 - 20^2 = \mathbf{O(n^4)}$
    - $0.8n + 224 = O(n)$
  - Em uma soma/subtração podemos nos focar na p**arcela dominante**:
    - $\mathbf{2^n} + 6n^3 = \mathbf{O(2^n)}$
    - $\mathbf{n!} - 3n^2 = \mathbf{O(n!)}$
    - $n \log n + 3\mathbf{n^2} = \mathbf{O(n^2)}$

4. Notação O

- **Exemplos Práticos:**
  - Um programa tem dois pedaços de código **A** e **B**, executados um a seguir ao outro, sendo que **A** corre em $O(n \log n)$ e **B** em $O(n^2)$.
    - O programa corre em $O(n^2)$, porque $n^2 > n \log n$
  - Um programa chama **n** vezes uma função $O(\log n)$, e em seguida volta a chamar novamente **n** vezes outra função $O(\log n)$
    - O programa corre em $O(n \log n)$
  - Um programa tem 5 ciclos, chamados sequencialmente, cada um deles com complexidade $O(n)$
    - O programa corre em $O(n)$
  - Um programa P_1 tem tempo de execução proporcional a $100 \times n \log n$. Um outro programa P_2 tem $2 \times n^2$. Qual é o programa mais eficiente?
    - P_1 é mais eficiente porque $n^2 > n \log n$. No entanto, para um n pequeno, P_2 é mais rápido e pode fazer sentido ter um programa que chama P_1 ou P_2 de acordo com o valor de **n**.

4. Notação O

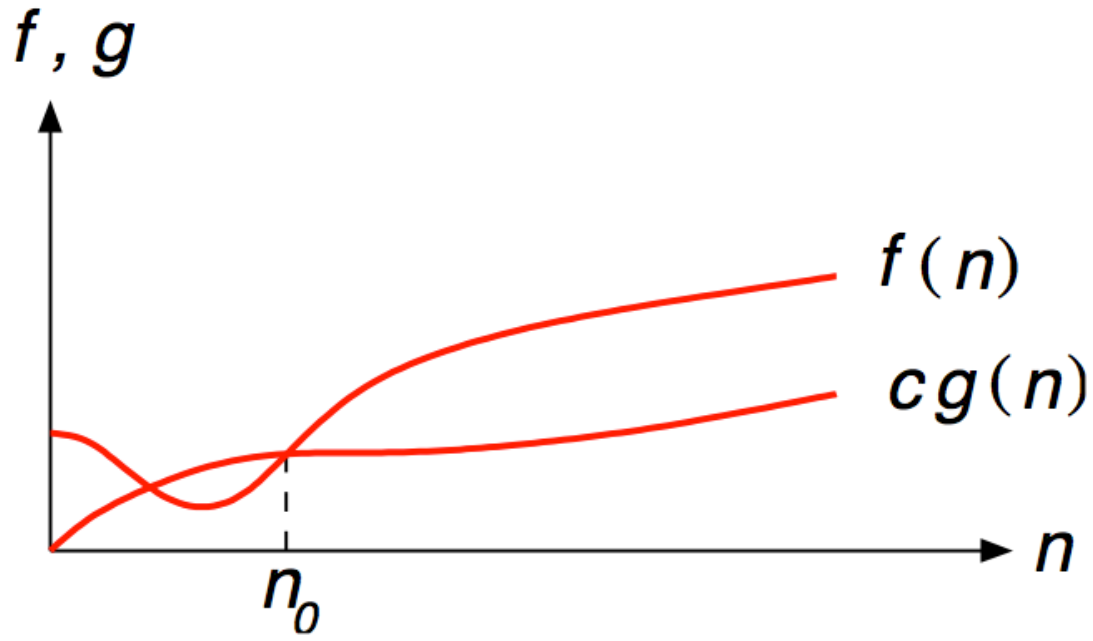
- **Exercício:**
  - Considere $f(n) = 3n^2 - 100n + 6$
    a) $f(n) = O(n^2)$?
    b) $f(n) = O(n^3)$?
    c) $f(n) = O(n)$?

4. Notação O

- **Exercício:**
  - Considere $f(n) = 3n^2 - 100n + 6$
    a) $f(n) = O(n^2)$? SIM
    b) $f(n) = O(n^3)$? SIM
    c) $f(n) = O(n)$? NÃO -> $f(n) \neq O(n)$

5. Notação Ω

- **f(n) = Ω(g(n))**
  - Significa que ***c* × *g(n)*** é um limite inferior de ***f(n)***



5. Notação Ω

- A notação **Ω** define um **limite inferior** para a função, por um fator constante.

- Escreve-se $f(n) = \Omega(g(n))$, se existirem constantes positivas **c** e $\mathbf{n_0}$ tais que para $\mathbf{n} \geq \mathbf{n_0}$, o valor de $f(n)$ é maior ou igual a $c \times g(n)$.

  - Pode-se dizer que $g(n)$ é um **limite assintótico inferior** (em inglês, asymptotically lower bound) para f(n)

$$f(n) = \Omega(g(n)) \text{ , } \exists\ c > 0 \text{ e } n_0 \mid 0 \leq c \times g(n) \leq f(n), \forall n \geq n_0$$

- Observe que a notação **Ω** define um conjunto de funções:

$$\Omega(g(n)) = \{ f: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists\ c > 0,\ n_0,\ 0 \leq c \times g(n) \leq f(n),\ \forall n \geq n_0 \}$$

5. Notação Ω

- Quando a notação **Ω** é usada para expressar o tempo de execução de um algoritmo no melhor caso, está se definindo também o limite (inferior) do tempo de execução desse algoritmo para todas as entradas.

- Por exemplo, o algoritmo de ordenação por inserção é ***Ω(n)*** no melhor caso.
  - O tempo de execução do algoritmo de ordenação por inserção é *Ω(n).*

- O que significa dizer que **"o tempo de execução"** (sem especificar se é para o pior caso, melhor caso, ou caso médio) é ***Ω(g(n))*** ?
  - O tempo de execução desse algoritmo é pelo menos uma constante vezes *g(n)* para valores suficientemente grandes de n.

5. Notação Ω

- Exemplos:

1. Para mostrar que $f(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer c = 1, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.

2. Seja $f(n) = n$ para **n** ímpar $(n \geq 1)$ e $f(n) = n^2/10$ para n par $(n \geq 0)$.
   - Neste caso $f(n)$ é $\Omega(n^2)$, bastando considerar $c = 1/10$ e $n = 0,2,4,6,..$

5. Notação Ω

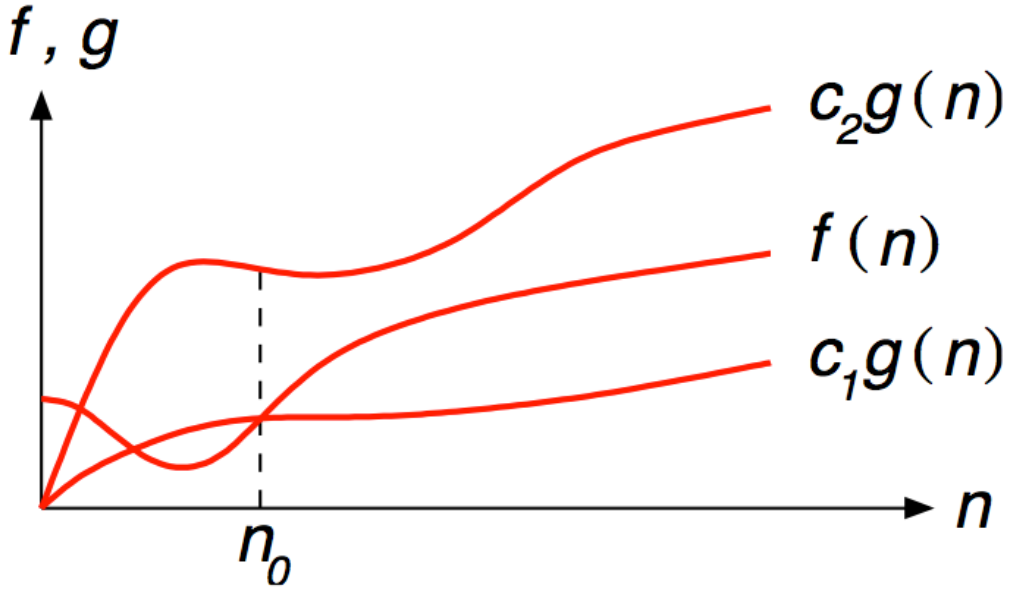
- **Exercício:**
  - Considere $f(n) = 3n^2 - 100n + 6$
    a) $f(n) = \Omega(n^2)$?
    b) $f(n) = \Omega(n^3)$?
    c) $f(n) = \Omega(n)$?

5. Notação Ω

- **Exercício:**
  - Considere $f(n) = 3n^2 - 100n + 6$
    a) $f(n) = \Omega(n^2)$? SIM
    b) $f(n) = \Omega(n^3)$? NÃO -> $f(n) \neq \Omega(n^3)$
    c) $f(n) = \Omega(n)$? SIM

6. Notação Θ

- **f(n) = Θ(g(n))**
  - Significa que $c_1 \times g(n)$ é um limite inferior de $f(n)$ e $c_2 \times g(n)$ é um limite superior de $f(n)$



6. Notação Θ

- A notação **Θ** limita a função por fatores constantes.
- Escreve-se $f(n) = \Theta(g(n))$, se existirem constantes positivas $\mathbf{c_1}$, $\mathbf{c_2}$ e $\mathbf{n_0}$ tais que para $\mathbf{n \geq n_0}$, o valor de $f(n)$ está sempre entre $c_1 \times g(n)$ e $c_2 \times g(n)$ inclusive.
  - Pode-se dizer que $g(n)$ é um **limite assintótico firme** (em inglês, asymptotically tight bound) para f(n)

$$f(n) = \Theta(g(n)) \text{ , } \exists\ c_1 > 0,\ c_2 > 0 \text{ e } n_0 \text{ /}$$
$$0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \text{ , } \forall n \geq n_0$$

- Observe que a notação **Θ** define um conjunto de funções:

$$\Theta(g(n)) = \{ f: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists\ c_1 > 0,\ c_2 > 0,\ n_0,$$
$$0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \text{ , } \forall n \geq n_0 \}$$

6. Notação Θ

- Exemplos:

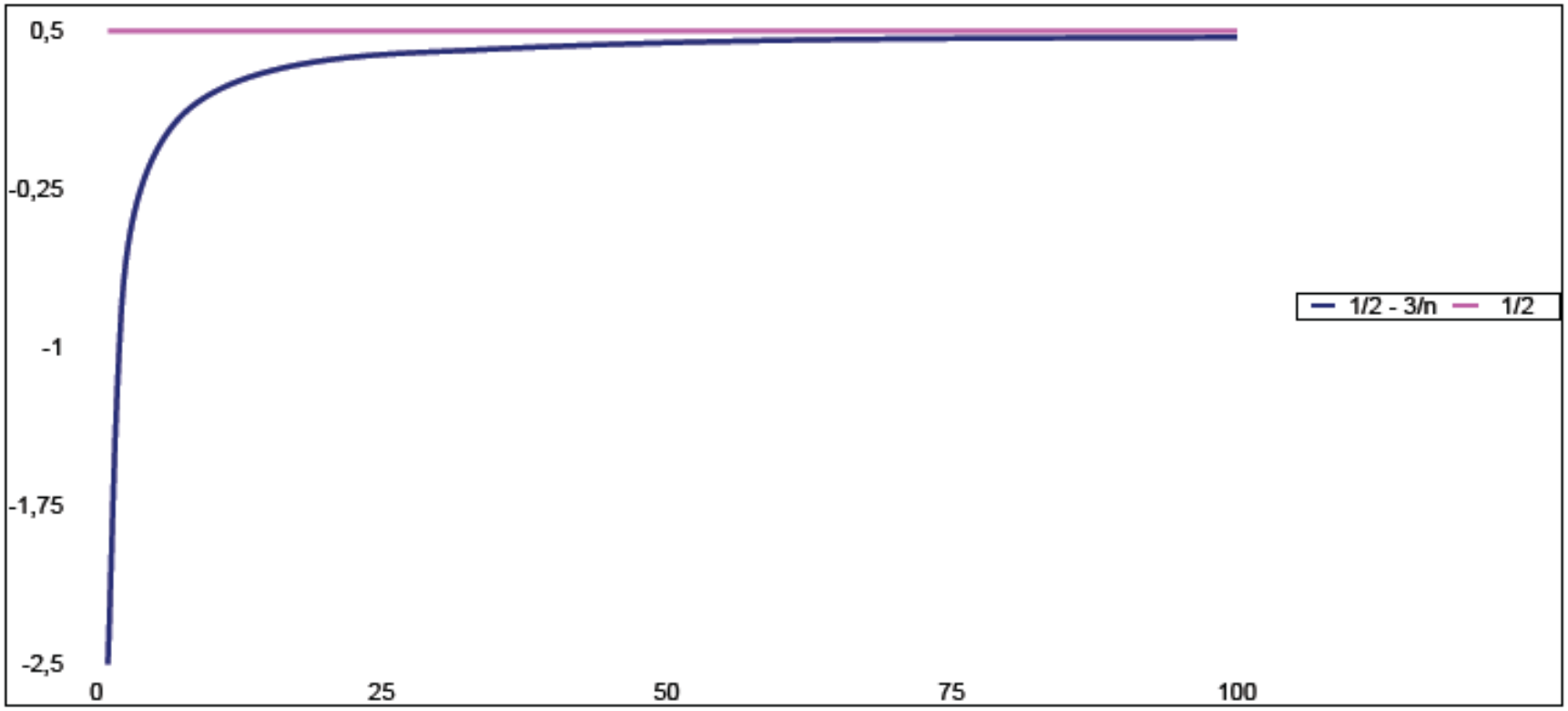
1. Mostre que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

   - Para provar esta afirmação, devemos achar constantes $c_1 > 0$, $c_2 > 0$, $n_0 > 0$, tais que: $\boldsymbol{c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2}$, para todo $n \geq n_0$.

   - Se dividirmos a expressão acima por $\boldsymbol{n^2}$, temos:

$$\boldsymbol{c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2}$$

6. Notação Θ



6. Notação Θ

- Exemplos:
  - A inequação mais a direita será sempre válida para qualquer valor de n $\geq$ 1 ao escolhermos $c_2 \geq \frac{1}{2}$
  - Da mesma forma, a inequação mais a esquerda será sempre válida para qualquer valor de n $\geq$ 7 ao escolhermos $c_1 \leq \frac{1}{14}$
  - Assim, ao escolhermos $c_1 \leq \frac{1}{14}$, $c_2 \leq \frac{1}{2}$ e $n_0 = 7$, podemos verificar que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
  - Note que existem outras escolhas para as constantes c_1 e c_2, mas o fato importante é que **a escolha existe.**
  - Note também que a escolha destas constantes depende da função $\frac{1}{2}n^2 - 3$
  - Uma função diferente pertencente a $\Theta(n^2)$ irá provavelmente requerer outras constantes

6. Notação Θ

- **Exercício:**
  - Considere $f(n) = 3n^2 - 100n + 6$
    a) $f(n) = \Theta(n^2)$?
    b) $f(n) = \Theta(n^3)$?
    c) $f(n) = \Theta(n)$?

6. Notação Θ

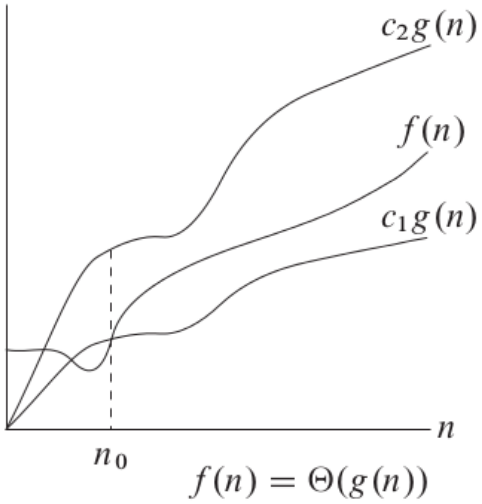
- **Exercício:**
  - Considere $f(n) = 3n^2 - 100n + 6$
    a) $f(n) = \Theta(n^2)$? SIM
    b) $f(n) = \Theta(n^3)$? NÃO -> $f(n) \neq \Theta(n^3)$
    c) $f(n) = \Theta(n)$? NÃO -> $f(n) \neq \Theta(n^3)$

  - $f(n) = \Theta(g(n))$ implica que $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

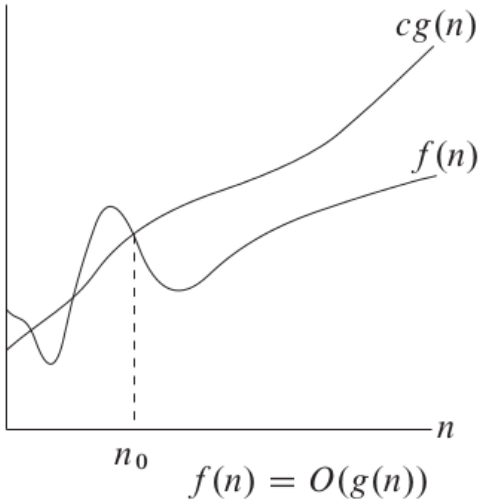
7. Outras Informações

- **Resumo das Notações**

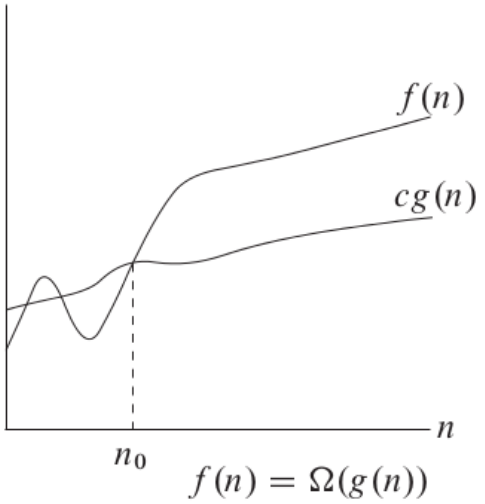
Θ



O



Ω



7.1. Propriedades das Notações

- **Reflexividade:**
  - $f(n) = O(f(n))$.
  - $f(n) = \Omega(f(n))$.
  - $f(n) = \Theta(f(n))$.
- **Simetria:**
  - $f(n) = \Theta(g(n))$ se, e somente se, $g(n) = \Theta(f(n))$.
- **Simetria Transposta:**
  - $f(n) = O(g(n))$ se, e somente se, $g(n) = \Omega(f(n))$.
- **Transitividade:**
  - Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$.
  - Se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$, então $f(n) = \Omega(h(n))$.
  - Se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = \Theta(h(n))$.

7. Outras Informações

- Quais as notações mais indicadas para expressar a complexidade de casos específicos de um algoritmo, do algoritmo de modo geral e da classe de algoritmos para o problema?

  - **Casos específicos:**
    - O ideal é a notação Θ, por ser um limite assintótico firme.
    - A notação O também é aceitável e bastante comum na literatura.
    - Embora possa teoricamente ser usada, a notação Ω é mais fraca neste caso e deve ser evitada para casos específicos.

  - **Algoritmo de forma geral:**
    - Se o algoritmo comporta-se de forma idêntica para qualquer entrada, a notação Θ é a mais precisa (lembre-se que $f(n)=\Theta(g(n)) \Rightarrow f(n)=O(g(n))$).
    - Se os casos melhor e pior são diferentes, a notação mais indicada é a O, já que estaremos interessados em um limite assintótico superior.
    - O pior caso do algoritmo deve ser a base da análise.

  - **Para uma classe de algoritmos:**
    - Neste caso estamos interessados no limite inferior para o problema e a notação deve ser a Ω.

7.2. Outras Notações

- Existem duas outras notações na análise assintótica de funções:
  1. **Notação o (“O” pequeno)**
  2. **Notação ω (ômega minúsculo)**

- Estas duas notações não são usadas normalmente, mas é importante saber seus conceitos e diferenças em relação às notações **O** e **Ω**, respectivamente.

7.2.1. Notação o

- O limite assintótico superior definido pela notação **O** pode ser assintoticamente firme ou não.
  - Por exemplo, o limite $2n^2 = O(n^2)$ é assintoticamente firme, mas o limite $2n = O(n^2)$ não é.

- A notação ***o*** é usada para definir um limite superior que **não é assintoticamente firme**.

- Formalmente a notação ***o*** é definida como:
  - $f(n) = o(g(n))$, *para qualquer* $c > 0$ e n_0 / $0 \leq f(n) < c \times g(n), \forall n \geq n_0$

- Exemplo:
  - $2n = o(n^2)$ mas $2n^2 \neq o(n^2)$

7.2.1. Notação o

- As definições das notações **O** e **o** são similares
  - A diferença principal é que em ***f(n) = o(g(n))***, a expressão $0 \leq f(n) < c \times g(n)$ é válida para todas constantes $c > 0$.

- Intuitivamente, a função ***f(n)*** tem um crescimento muito menor que ***g(n)*** quando ***n*** tende para infinito.
  - Isto pode ser expresso da seguinte forma:

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 0$$

- Alguns autores usam este limite como a definição de ***o***

7.2.2. Notação ω

- Por analogia, a notação **ω** está relacionada com a notação **Ω** da mesma forma que a notação **o** está relacionada com a notação **O**
- Formalmente a notação **ω** é definida como:
  - $f(n) = \omega(g(n))$, *para qualquer* $c > 0$ e $n_0 \mid 0 \leq c \times g(n) < f(n), \forall n \geq n_0$
- Por exemplo, $\frac{n^2}{2} = \omega(n)$, mas $\frac{n^2}{2} \neq \omega(n^2)$
- A relação ***f(n) = ω(g(n))*** implica em:

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = \infty$$

se o limite existir!

7.3. Classes de Comportamento Assintótico

- Se **f** é uma função de complexidade para um algoritmo **F**, então **O(f)** é considerada a **complexidade assintótica** ou o **comportamento assintótico** do algoritmo **F**.

- A relação de dominação assintótica permite comparar funções de complexidade.

- Entretanto, se as funções **f** e **g** dominam assintoticamente **uma a outra**, então os algoritmos associados são equivalentes.
  - Nestes casos, o comportamento assintótico não serve para comparar os algoritmos.

- Por exemplo, considere dois algoritmos **F** e **G** aplicados à mesma classe de problemas, sendo que **F** leva **três vezes** o tempo de **G** ao serem executados, isto é, $f(n) = 3g(n)$, sendo que **O(f(n)) = O(g(n))**.
  - Logo, o comportamento assintótico não serve para comparar os algoritmos **F** e **G**, porque eles diferem apenas por uma **constante**.

7.4. Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.

- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$
  - Porém, as constantes de proporcionalidade podem alterar esta consideração.

- Exemplo: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?
  - Depende do tamanho do problema
    - Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possui tempo $100n$
    - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$
    - Entretanto, quando **n** cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$

7.5. Classes de Comportamento Assintótico

1. **Complexidade constante** ← f(n) = O(1)

   - O uso do algoritmo independe do tamanho de **n**
   - As instruções do algoritmo são executadas um número fixo de vezes

7.5. Classes de Comportamento Assintótico

2. **Complexidade Logarítmica** ← **f(n) = O(log n)**

   - Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.
   - Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande.
   - Supondo que a base do logaritmo seja 2:
     - Para n = 1.000, $\log_2 \approx 10$
     - Para n = 1.000.000, $\log_2 \approx 20$
   - Exemplo:
     - Algoritmo de pesquisa binária

7.5. Classes de Comportamento Assintótico

3. **Complexidade Linear** ← **f(n) = O(n)**

   - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
   - Esta é a melhor situação possível para um algoritmo que tem que processar/produzir n elementos de entrada/saída
   - Cada vez que n dobra de tamanho, o tempo de execução também dobra
   - Exemplo:
     - Algoritmo de pesquisa sequencial

7.5. Classes de Comportamento Assintótico

4. **Complexidade Linear Logarítmica** ← f(n) = O(n log n)

   - Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções

   - Caso típico dos algoritmos baseados no paradigma divisão-e-conquista.

   - Supondo que a base do logaritmo seja 2:
     - Para n = 1.000.000, n $\log_2 \approx$ 20.000.000
     - Para n = 2.000.000, n $\log_2 \approx$ 42.000.000

   - Exemplo:
     - Algoritmo de ordenação MergeSort

7.5. Classes de Comportamento Assintótico

5. **Complexidade Quadrática** ← $f(n) = O(n^2)$

   - Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro do outro
   - Para **n** = 1.000, o número de operações é da ordem de 1.000.000
   - Sempre que **n** dobra o tempo de execução é multiplicado por 4
   - Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos
   - Exemplos:
     - Algoritmos de ordenação simples como seleção e inserção

7.5. Classes de Comportamento Assintótico

6. **Complexidade Cúbica** ← $f(n) = O(n^3)$

   - Algoritmos desta ordem de complexidade geralmente são úteis apenas para resolver problemas relativamente pequenos
   - Para **n = 100**, o número de operações é da ordem de **1.000.000**
   - Sempre que **n** dobra o tempo de execução é multiplicado por 8
   - Exemplo:
     - Algoritmo para multiplicação de matrizes

7.5. Classes de Comportamento Assintótico

7. **Complexidade Exponencial** ← $f(n) = O(2^n)$

   - Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático
   - Eles ocorrem na solução de problemas quando se usa a **força bruta** para resolvê-los
   - Para **n = 20**, o tempo de execução é cerca de **1.000.000**
   - Sempre que **n** dobra o tempo de execução fica elevado ao quadrado

7.5. Classes de Comportamento Assintótico

8. **Complexidade Fatorial** ← **f(n) = O(n!)**

   - Um algoritmo de complexidade **O(n!)** é dito ter complexidade fatorial (ou até mesmo exponencial), apesar de **O(n!)** ter comportamento muito pior do que **$O(2^n)$**

   - Geralmente ocorrem quando se usa **força bruta** na solução do problema

   - Considerando:
     - **n = 20**, temos que **20! = 2.432.902.008.176.640.000**, um número com **19 dígitos**
     - **n = 40** temos um número com **48 dígitos**

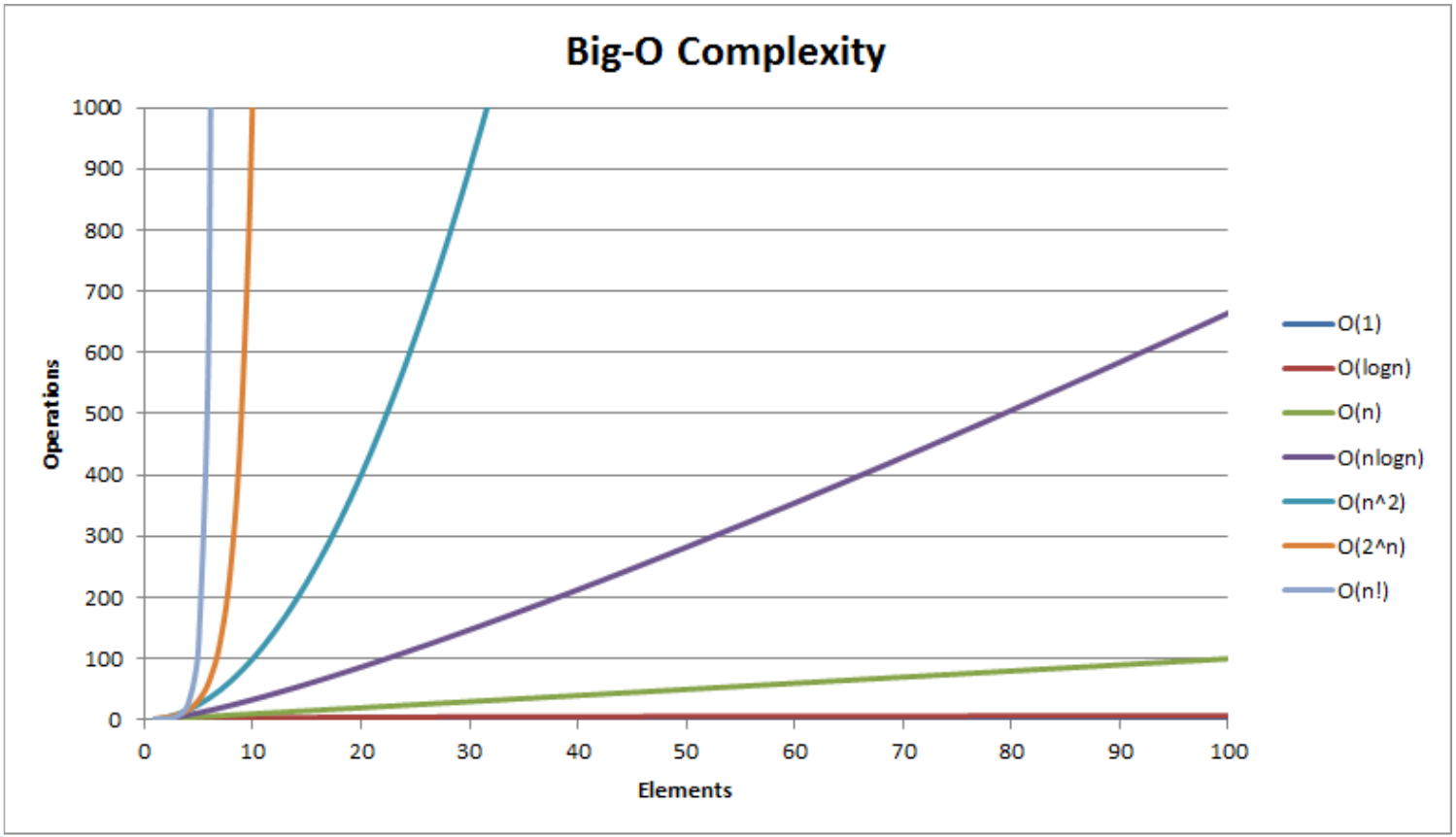
7.5. Classes de Comportamento Assintótico

- **Crescimento Assintótico**

Função	Nome	Exemplos
1	constante	Somar dois números
log n	logarítmica	Pesquisa binária, inserir um número em uma heap
n	linear	1 ciclo para buscar o valor máximo em um vetor
n log n	linearítimica	Ordenação (merge sort, heap sort)
n^2	quadrática	2 ciclos (bubble sort, selection sort)
n^3	cúbica	3 ciclos (Floyd-Warshall)
2^n	exponencial	Pesquisa exaustiva (subconjuntos)
n!	fatorial	Todas as permutações

7.5. Classes de Comportamento Assintótico

- Crescimento Assintótico



7.5. Classes de Comportamento Assintótico

- **Crescimento Assintótico**

$f(n)$	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0025 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0,316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1,0 s	17,9 min	12,7 dias	35,7 anos	366 séc
3^n	0,059 s	58 min	6,5 anos	3855 séc	10^8 séc	10^{13} séc

7.6. Revisão de Matemática

- Propriedades de Logaritmos:
  - $\log_b xy = \log_b x + \log_b y$
  - $\log_b \frac{x}{y} = \log_b x - \log_b y$
  - $\log_b x^a = a \log_b x$
  - $b^{\log_c a} = a^{\log_c b}$
  - $\log_b a = \frac{\log_c a}{\log_c b}$

  - $\log_a b = x \leftrightarrow a^x = b$

7.6. Revisão de Matemática

- Propriedades de Expoentes:
  - $a^{b+c} = a^b \times a^c$
  - $\left(a^b\right)^c = a^{bc}$
  - $\frac{a^b}{a^c} = a^{b-c}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c \times \log_a b}$

7.7. Regras para Cálculo de Tempo

- Repetições:
  - O tempo de execução de uma repetição é o tempo dos comandos dentro da repetição (incluindo testes) multiplicado pelo número de vezes que é executada.

7.7. Regras para Cálculo de Tempo

- Repetições aninhadas:
  - A análise é feita de dentro para fora
  - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições.
  - O exemplo abaixo é $O(n^2)$

```
para i := 0 até n faça
   para j := 0 até n faça
      faça k := k+1;
```

7.7. Regras para Cálculo de Tempo

- Comandos consecutivos
  - É a soma dos tempos de cada um bloco, o que pode significar o máximo entre eles.
  - O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser $O(n)$

```
para i := 0 até n faça
   faça k := 0;

para i := 0 até n faça
   para j := 0 até n faça
      faça k := k+1;
```

7.7. Regras para Cálculo de Tempo

- Se... então... senão
  - Para uma cláusula condicional, o tempo de execução nunca é maior do que o **tempo do teste** mais o tempo do **maior** entre os comandos relativos ao **então** e os comandos relativos ao **senão**
  - O exemplo abaixo é *O(n)*

```
se i < j
então i := i+1
senão para k := 1 até n faça
   i := i*k;
```

7.7. Regras para Cálculo de Tempo

- Chamadas a sub-rotinas
  - Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou.

7.7. Regras para Cálculo de Tempo

- Sub-rotinas recursivas
  - **Análise de recorrência**
    - Recorrência: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores
  - **Caso típico**: algoritmos de dividir-e-conquistar, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

7.7. Regras para Cálculo de Tempo

- Sub-rotinas recursivas
  - Exemplo:

```
int fat (int n) {

   if (n == 1)
      return 1;

   else
      return n * fat(n-1);
}
```

$$
\begin{aligned}
T(n) &= c+T(n-1) \\
&= c+(c+T(n-2)) = 2c+T(n-2) \\
&= \ldots \\
&= nc + T(1) \\
&= O(n)
\end{aligned}
$$

7.8. Exemplo

	Custo	Nº Vezes
`int hasDuplicate(int* vet, int n){`	C_1	1
`  int i, j;`	C_2	1
`  int duplicate = 0;`	C_3	1
`  for (i = 0; i < n; i++){`	C_4	n+1
`    for (j = 0; j < n; j++){`	C_5	n x (n+1)
`      if (i != j && A[i] == A[j])`	C_6	n x n
`        return 1;`	C_7	1
`    }`	0	–
`  }`	0	–
`  return 0;`	C_8	1
`}`	0	–

$$T(n) = C_1+C_2+C_3+C_4(n+1)+C_5(n^2+n)+ C_6n^2+C_7+C_8 = O(n^2)$$

7.8. Exemplo

- **Operações com a notação O**

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \quad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

8. Referências

- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. **Introduction to Algorithms**. 3a Ed. Cambridge: MIT Press and McGraw-Hill, 2009. ISBN: 0-262-03384-4.

- Levitin, Anany. **Introduction to the Design and Analysis of Algorithms**. 3a Ed. Nova Jérsei: Addison- Wesley, 2012. ISBN: 0-13-231681-1