Análise de Complexidade

Estrutura de Dados II

Prof. João Dallyson Sousa de Almeida
Núcleo de Computação Aplicada NCA - UFMA
Dep. De Informática - Universidade Federal do Maranhão

Apresentação

- Ementa
  - Algoritmos de ordenação e busca.
  - Árvore de busca multidirecional balanceada.
  - Hashing. Noções de organização de arquivos.
  - Noções de grafos: conceitos, coloração, árvores geradoras..
  - Algoritmos em grafos: caminho mínimo, fluxo máximo e outros.

- Bibliografia: básica
  - CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Editora Campus, 2002
  - Algorithms 4th edition by R. Sedgewick and K. Wayne, Addison-Wesley Professional, 2011, ISBN 0-321-57351-X
  - Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Cengage Learning, 2004.

- Bibliografia: complementar
  - TENENBAUM, Aaron; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. Estruturas de dados usando C. São Paulo: Makron Books, 1995. ISBN: 9788534603485
  - ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: Algoritmos, análise da complexidade e implementações em Java e C/C++. Pearson Prentice Hall, 2010
  - DROZDEK, Adam. Adam Drozdek. Data Structures and Algorithms in Java. 2. Cengage Learning. 2004. 2. Cengage Learning. 2004
  - GOODRICH, Michael T. Estruturas de dados e algoritmos em java. 4 ED. Porto Alegre: Bookman, 2007. 600.
  - SKIENA, Steven S.. The Algorithm Design Manual. 2. Springer-Verlag. 2008

Motivação: Custos

- Infelizmente os computadores têm recursos limitados!
  - Recurso: poder de processamento (tempo)
  - Recurso :armazenagem de dados (memória)

- Dois algoritmos distintos que realizam a mesma tarefa podem diferenciar brutalmente em relação aos custos em tempo e memória!

Motivação: exemplo

- Seja dois métodos de ordenação:
  - Ordenação por inserção:
    - Custo em tempo: $c_1 n^2$ para ordenar n números
  - Ordenação por intercalação (Merge Sort):
    - Custo em tempo: $c_2 n \log_2 n$ para ordenar para ordenar n números
- Suponha dois computadores:
  - Computador A:
    - Executa 1.000.000.000 de instruções por segundo
  - Computador B:
    - Executa 10.000.000 de instruções por segundo

Motivação: exemplo cont...

- O melhor programador do mundo implementa a ordenação por inserção em código de máquina no computador A
- Um programador mediano implementa a ordenação por intercalação em linguagem de alto-nível no computador B
- Tempo em cada computador (ordenar um milhão de números)
  - Computador A ($c_1 = 2$)

$$\frac{2 \cdot (10^6)^2 \ instruções}{10^9 \ instruções / segundo} = 2.000 \ segundos$$

  - Computador B ($c_2 = 50$)

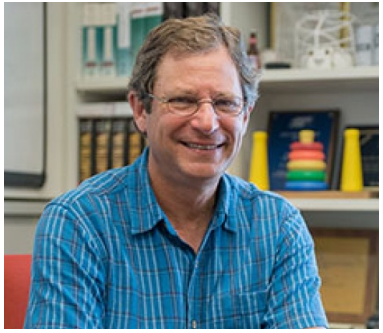
$$\frac{50 \cdot 10^6 \log_2 10^6 \ instruções}{10^7 \ instruções / segundo} \approx 100 \ segundos$$

Motivação: exemplo cont...

- Desta forma:
  - Mesmo utilizando um compilador fraco, o computador B funciona 20 vezes mais rápido que o computador A!
  - Este exemplo mostra que a escolha do algoritmo pode ser bem mais crítica do que a escolha do Hardware e da linguagem e/ou experiência do programador!
- Portanto:
  - Tanto os algoritmos quanto o Hardware constituem uma tecnologia!
  - O desempenho total do sistema depende da escolha correta de ambos!

O que é a análise de algoritmos

- Segundo Cormen (2002), é a previsão dos recursos de que o algoritmo necessitará.
  - Memória.
  - Largura de banda de comunicação.
  - *Hardware* de computação.
  - Tempo de computação.



Thomas H. Cormem
Prof. Emérito de Ciência da Computação
Dartmouth College

Como escolher um algoritmo?

- **Tempo de Processamento?**
  - Um algoritmo que realiza uma tarefa em 4 horas é melhor que outro que realiza em 4 dias

- **Quantidade de Memória necessária?**
  - Um algoritmo que usa 1MB de memória RAM é melhor que outro que usa 1GB

Complexidade Computacional

- **Termo criado por Juris Hartmanis e Richard Stearns (1965)**

- **Relação entre o tamanho do problema e seu consumo de tempo e espaço durante a execução**

Tempo de Processamento

- **Medir o tempo gasto por um algoritmo**
  - Não é a melhor opção
  - Depende do compilador
    - Pode preferir algumas construções ou otimizar melhor
  - Depende do hardware
    - GPU vs CPU, desktop vs. Smartphone.

Estudar o número de vezes que operações são executadas	

Tipos de Problemas

- **Analisando classes de algoritmos.**
  - Qual é o algoritmo de menor custo possível para resolver um problema particular?
  - Toda 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.

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

O modelo RAM de computação

- **Os algoritmos** são uma parte importante e durável da ciência da computação, porque eles **podem ser estudados em uma máquina / linguagem de forma independente.**
- Modelo RAM de computação:
  - Cada operação simples (+,-,=,IF,chamada) leva 1 passo.
  - Loops e subrotinas não são operações simples. Eles dependem do tamanho do dado e do conteúdo de uma sub-rotina. “Ordenação” não é uma operação de 1 passo simples.
  - Cada acesso à memória custa exatamente 1 passo.

Método do Custo por meio de um Modelo Matemático

- Usa um modelo matemático baseado em um computador idealizado
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- Recomenda-se ignorar o custo de algumas das operações e considerar apenas as operações mais significativas (Turing, 1947).
- 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.

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.
- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente.

Exemplo: Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteirosv[0..n – 1], n

```
int vmax(int *vec, int n) {          -
        int i;                       -
        int max = vec[0];            1
        for(i = 1; i < n; i++) {     n-1
                if(vec[i] > max) {   n-1
                        max = vec[i];  A < n-1
                }                    n-1
        }                            n-1
        return max;                  1
}
```

Exemplo: Maior Elemento (2)

- Seja f uma função de complexidade tal que f(n) é o número de comparações entre os elementos de v, se v contiver n elementos.
- Para encontrar o maior, é necessário mostrar que cada um dos n-1 elementos é maior que algum outro.
- Logo f(n) = n – 1, para n > 0.
- Esse algoritmo é ótimo.

Tamanho da Entrada de Dados

- 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 do método 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.

Melhor Caso, Pior Caso e Caso Médio

- **Melhor caso: menor tempo de execução** sobre todas as entradas de tamanho n.
- **Pior caso: 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.
- Na análise do caso 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.

Exemplo - Registros de um Arquivo

- Considere o problema de acessar os **registros de um arquivo.** Cada registro contém uma **chave única que é** utilizada para recuperar registros do arquivo.

- O problema: dada uma chave qualquer, localize o registro que contenha esta chave.
  - O algoritmo de pesquisa mais simples é o que faz a **pesquisa sequencial.**

Exemplo - Registros de um Arquivo (2)

- Seja f uma função de complexidade tal que f(n) é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - **melhor caso:** f(n) = 1 (registro procurado é o primeiro consultado);
  - **pior caso: f(n) = n (registro procurado é o** último consultado ou não está presente no arquivo);
  - **caso médio: f(n) = (n + 1)/2.**

Exemplo - Maior e Menor Elemento

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros v[0..n − 1], n 1.

- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento.

- O vetor maxMin definido localmente no método maxMin1 é utilizado para retornar nas posições 0 e 1 o maior e o menor elemento do vetor v, respectivamente.

Exemplo - Maior e Menor Elemento (2)

```
package cap1;
public class MaxMin1 {
  public static int [] maxMin1 (int v[], int n) {
    int max = v[0], min = v[0];
    for (int i = 1; i < n; i++) {
      if (v[i] > max) max = v[i];
      if (v[i] < min) min = v[i];
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```

melhor caso: $f(n) = 2(n-1)$
pior caso: $f(n) = 2(n-1)$
caso médio: $f(n) = 2(n-1)$

Exemplo - Maior e Menor Elemento (3)

- Seja f(n) o número de comparações entre os elementos de v, se v contiver n elementos.
- Logo f(n) = 2(n – 1), para n > 0, para o melhor caso, pior caso e caso médio.
- MaxMin1 pode ser facilmente melhorado: a comparação v[i] < min só é necessária quando a comparação v[i] > max é falsa.
- A seguir, apresentamos essa versão melhorada

Exemplo - Maior e Menor Elemento (4)

```
package cap1;
public class MaxMin2 {
  public static int [] maxMin2 (int v[], int n) {
    int max = v[0], min = v[0];
    for (int i = 1; i < n; i++) {
      if (v[i] > max) max = v[i];
      else if (v[i] < min) min = v[i];
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```

melhor caso:
(crescente)
$f(n) = n-1$

pior caso:
(decrescente)
$f(n) = 2(n-1)$

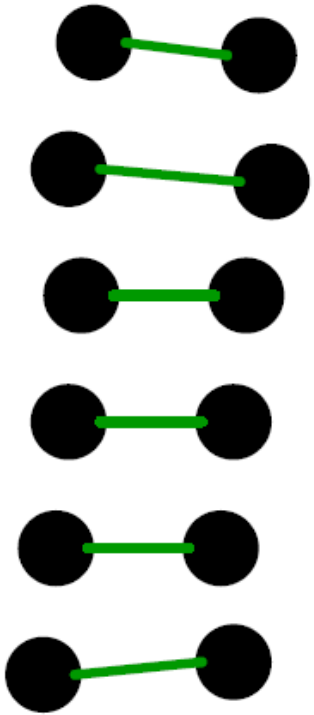
caso médio:
(aleatório)
$f(n) > 3(n-1)/2$

Exemplo - Maior e Menor Elemento (5)

- Para a nova implementação temos:
  - **melhor caso: f(n) = n – 1 (**quando os elementos estão em ordem crescente);
  - **pior caso: f(n) = 2(n – 1) (**quando os elementos estão em ordem decrescente);
  - **caso médio: f(n) = 3n/2 – 3/2.**
- No caso médio, v[i] é maior do que max a metade das vezes.

$$f(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}, \quad \text{para } n > 0.$$

Ex. Maior Menor.

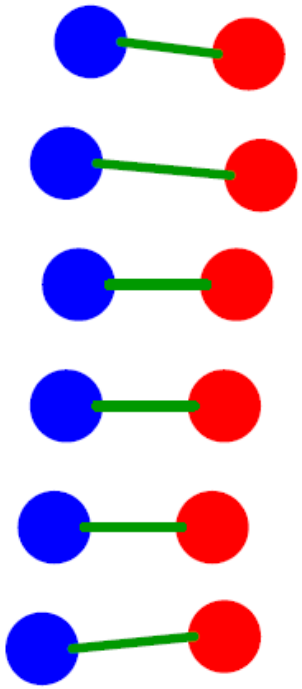


- Comparar elementos par-a-par
  - Custo: n/2 comparações

Mantra:

É possível fazer melhor?

Ex. Maior Menor. Dá para fazer melhor?



- Comparar elementos par-a-par
  - Custo: n/2 comparações
- Elementos vermelhos são maiores que os azuis
- Encontrar o máximo entre os elementos vermelhos
  - Custo: n/2 comparações
- Encontrar o mínimo entre os elementos azuis
  - Custo: n/2 comparações

Ex. Maior Menor. Dá para fazer melhor?

```
        -   void minmax3(int *vec, int n, int *min, int *max) {
        -      int i;
        1      int *min = INT_MAX;
        1      int *max = INT_MIN;
      n/2      for(i = 0; i < n; i += 2) {
      n/2         if(vec[i] < vec[i+1]) {
      n/4            a = i; v = i+1;
        -         } else {
      n/4            a = i+1; v = i;
        -         }
      n/2         if(vec[a] < *min)
A < n/2              *min = vec[a];
      n/2         if(vec[v] > *max)
B < n/2              *max = vec[v];
        -      }
        -   }
```

melhor caso:
$f(n) = 3n/2$
pior caso:
$f(n) = 3n/2$
caso médio:
$f(n) = 3n/2$

Algoritmo ótimo

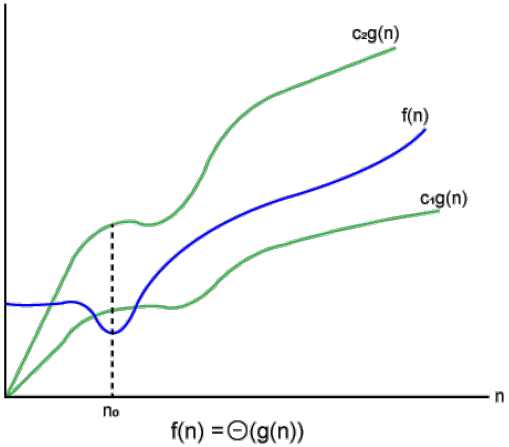
Ex. Maior Menor, comparação

Os Três Algoritmos	*f (n)*		
	Melhor caso	Pior caso	Caso médio
MaxMin1	2(n-1)	2(n-1)	2(n-1)
MaxMin2	$n - 1$	2(n-1)	*3n/2-3/2*
MaxMin3	*3n/2-2*	*3n/2-2*	*3n/2-2*

FONTE: [ZIVIANE, 2006]

Análise exata é difícil!

- Analisar o Melhor, o pior, e o caso médio são difíceis de tratar com precisão porque os detalhes são muito complicados.
- É fácil falar sobre limites superiores e inferiores da função. Notação assintótica (O, Θ, Ω) nos permite lidar com funções de complexidade



FONTE: https://www.javatpoint.com/data-structure-asymptotic-analysis

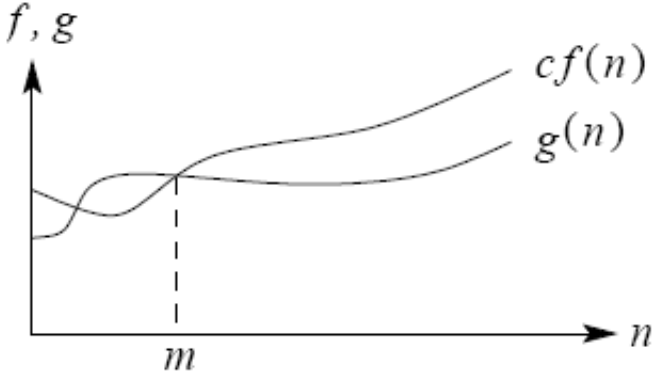
Comportamento assintótico

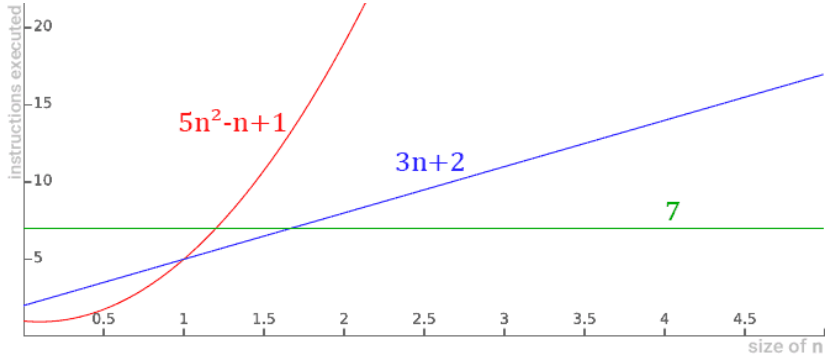
- Para valores suficientemente pequenos de *n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes*
  - Escolha de um algoritmo não é um problema crítico
- Logo, analisamos algoritmos para grandes valores de *n*
  - Estudamos o comportamento assintótico das funções de complexidade de um programa (comportamento pra grandes valores de *n)*

Dominação 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 m tais que, para n>=m, temos |g(n)| <= c × |f(n)|.

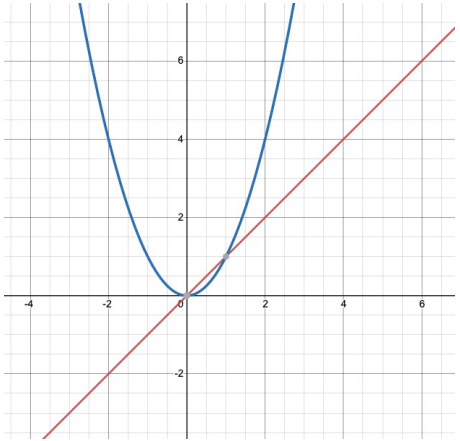
Dominação assintótica





Exemplos

- $f(n) = n$, $g(n) = -n^2$
  - f(n) não domina assintoticamente g(n)
  - c=1, m=0
  - |f(n)| <= 1|g(n)| para todo n >= m = 0



Verificação da equação $|n| \leq c \cdot |-n^2|$

n	$\lvert n\rvert \leq c \cdot \lvert -n^2\rvert$ $(c = 1)$
1	$1 \leq 1$
2	$2 \leq 4$
3	$3 \leq 9$
4	$4 \leq 16$
...	...

Exemplos

- Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$.
- As funções g(n) e f(n) dominam assintoticamente uma a outra. Qual a prova?

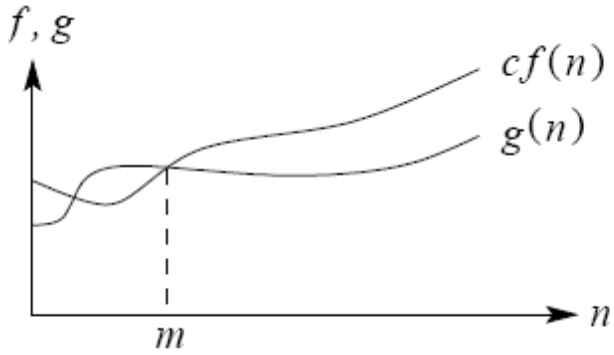
Notação O (ômicron maiúscula)

- Escrevemos g(n) = *O*(f(n)) para expressar que f(n) domina assintoticamente g(n). Lê-se g(n) é da ordem de no máximo f(n).

- Exemplo: quando dizemos que o tempo de execução T(n) de um programa é *O*(n^2), significa que existem constantes c e m tais que, para valores de n >= *m*, T(n) <= *c*n^2.

Notação O

- Exemplo gráfico de dominação assintótica que ilustra a notação O.



- O valor da constante m mostrado é o menor valor possível, mas qualquer valor maior também é válido.
- A notação O nos dá um limite superior assintótico

Exemplos de Notação O

- Exemplo: $g(n) = (n + 1)^2$.

- Exemplo: $g(n) = n$ e $f(n) = n^2$

n>=m, temos |g(n)| <= c × |f(n)|

Exemplos de Notação O

- Exemplo: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.

Operações com a Notação O

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

Exemplo

- Regra da soma O(f(n)) + O(g(n)).
  - Suponha três trechos cujos tempos de execução são O(n), $O(n^2)$ e O(n log n).
  - O tempo de execução dos dois primeiros trechos é $O(max(n, n^2))$, que é $O(n^2)$.
  - O tempo de execução de todos os três trechos é então $O(max(n^2, n \log n))$, que é $O(n^2)$.

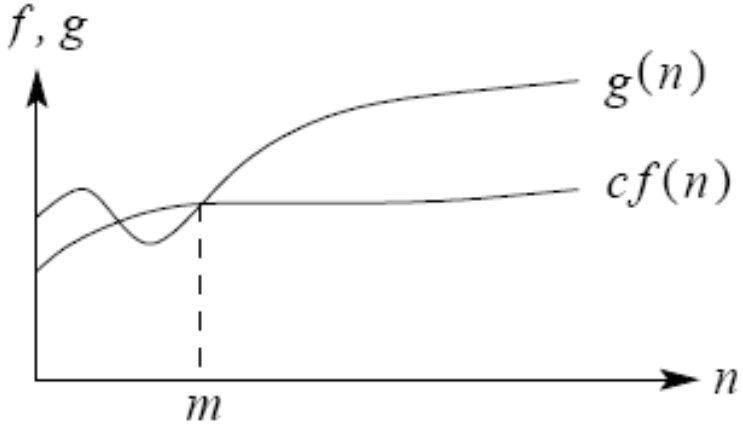
Notação Ω (Ômega)

- Especifica um limite inferior para g(n).
- Definição: Uma função g(n) é Ω(f(n)) se existirem duas constantes c e m tais que g(n) >= cf(n), para todo n >= m.
- **Exemplo:** Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer c = 1, e então $3n^3 + 2n^2 >= n^3$ para n >= 0.

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

Exemplo gráfico para a notação Ω (ômega)

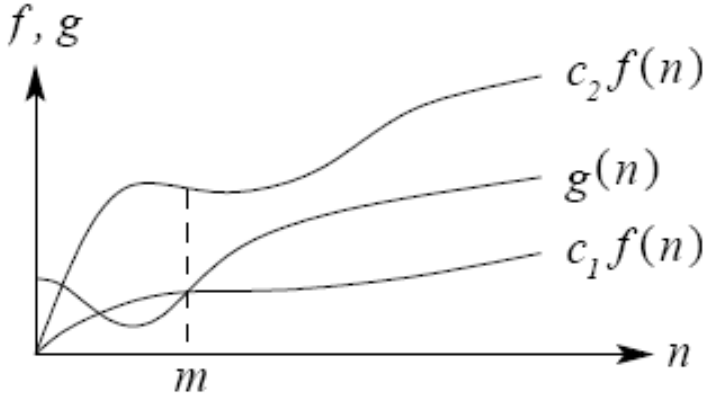
- Uma função $g(n)$ *é* $\Omega(f(n))$ *se* $g(n)$ *domina assintoticamente* $f(n)$
- Notação O denota um limite superior e a notação Ω denota um limite inferior



Para todos os valores à direita de m, o valor de g(n) está sobre ou acima do valor de cf(n).

Notação θ

- Definição: Uma função g(n) é θ(f(n)) se existirem constantes positivas c1, c2 e m tais que 0 <= c1f(n) <= g(n) <= c2f(n), para todo n >= m.
- Definição equivalente: $g(n) = \theta(f(n))$ *se* $g(n) = O(n)$ *e* $g(n) = \Omega(f(n))$



Notação θ

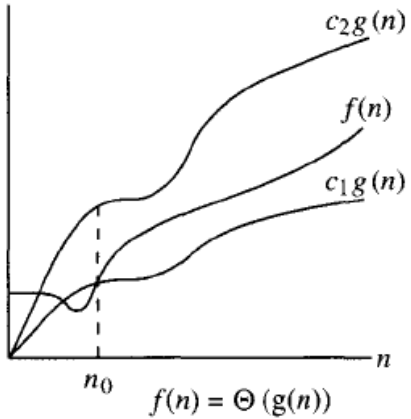
- Dizemos que g(n) = θ(f(n)) se existirem constantes c1, c2 e m tais que, para todo n >= m, o valor de g(n) está sobre ou acima de c1f(n) e sobre ou abaixo de c2f(n).
- Isto é, para todo n >= m, a função g(n) é igual a f(n) a menos de uma constante.
- Neste caso, f(n) é um **limite assintótico firme.**

Exemplo de Notação θ

- Seja $g(n) = n^2/3 - 2n$. Vamos mostrar que $g(n) = \theta(n^2)$.
- Temos de obter constantes c1, c2 e m tais que $c1n^2 <= (1/3)n^2 - 2n <= c2n^2$ para todo $n >= m$.

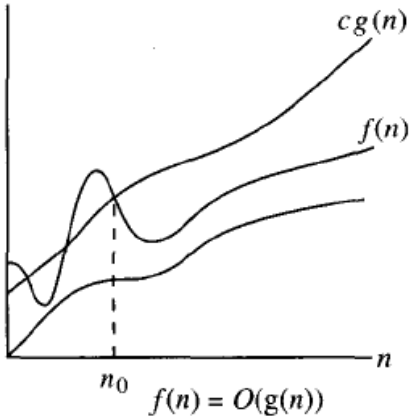
  - Dividindo por n^2 leva a $c1 <= 1/3 - 2/n <= c2$
  - O lado direito da desigualdade será sempre válido para qualquer valor de $n >= 1$ quando escolhemos $c2 >= 1/3$.
  - Escolhendo $c1 <= 1/21$, o lado esquerdo da desigualdade será válido para qualquer valor de $n >= 7$.
  - Logo, escolhendo $c1 = 1/21$, $c2 = 1/3$ e $m = 7$, verifica-se que $n^2/3 - 2n = (n^2)$.
  - Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.

Notação



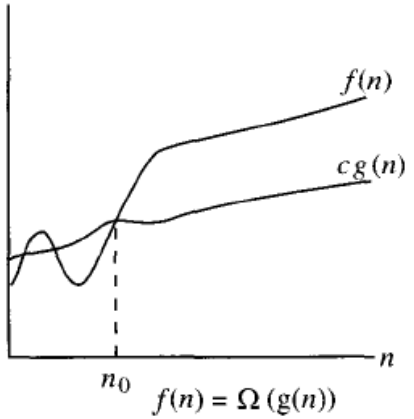
$f(n) = \Theta(g(n))$

(a)



$f(n) = O(g(n))$

(b)



$f(n) = \Omega(g(n))$

(c)

Usando limites para comparar ordem de crescimento

- Em vez de procurar por constantes **c e m** que satisfaçam uma inequação, podemos determinar diretamente se uma função é da ordem de outra verificando se a seguinte igualdade é satisfeita:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = k \text{, onde k} \in \Re.$$

Usando limites para comparar ordem de crescimento

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 \\ c \\ \infty \end{cases}$$

- t(n) tem ordem de crescimento menor que g(n)
- t(n) tem a mesma ordem de crescimento de g(n)
- t (n) tem ordem de crescimento maior que g(n)

Usando limites para comparar ordem de crescimento

Ex: comparar a ordem de crescimento de

(1/2)n(n-1) e n^2

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2} \lim_{n\to\infty} (1 - \frac{1}{n}) = \frac{1}{2}.$$

Como o limite é igual a uma constante positiva, as funções tem a mesma ordem de crescimento

$$\frac{1}{2}n(n-1) \in \Theta(n^2).$$

Exercício

- [Poscomp 2003]

37. Qual é o número mínimo de comparações necessário para encontrar o menor elemento de um conjunto qualquer não ordenado de n elementos?

(a) 1
(b) $n - 1$
(c) n
(d) $n + 1$
(e) $n \log n$

Exercicio POSCOMP 2015

Sejam $T_1(n) = 100 \cdot n + 15$, $T_2(n) = 10 \cdot n^2 + 2 \cdot n$ e $T_3(n) = 0{,}5 \cdot n^3 + n^2 + 3$ as equações que descrevem a complexidade de tempo dos algoritmos Alg1, Alg2 e Alg3, respectivamente, para entradas de tamanho n. A respeito da ordem de complexidade desses algoritmos, pode-se concluir que

(A) as complexidades assintóticas de Alg1, Alg2 e Alg3 estão, respectivamente, em $O(n)$, $O(n^2)$ e $O(n^3)$.

(B) as complexidades assintóticas de Alg1, Alg2 e Alg3 estão, respectivamente, em $O(n)$, $O(n^2)$ e $O(n^2)$.

(C) as complexidades assintóticas de Alg1, Alg2 e Alg3 estão, respectivamente, em $O(100)$, $O(10)$ e $O(0{,}5)$.

(D) Alg2 e Alg3 pertencem às mesmas classes de complexidade assintótica.

(E) Alg1 e Alg2 pertencem às mesmas classes de complexidade assintótica.

Exercício Poscomp 2017

QUESTÃO 21 – A análise de algoritmos que estabelece um limite superior para o tempo de execução de qualquer entrada é denominada análise

A) do melhor caso.
B) do caso médio.
C) do pior caso.
D) da ordem de crescimento.
E) do tamanho da entrada.

Exercício Poscomp 2022

QUESTÃO 22 – Considere as funções a seguir:

$$f1(n) = O(n)$$
$$f2(n) = O(n!)$$
$$f3(n) = O(2^n)$$
$$f4(n) = O(n^2)$$

A ordem dessas funções, por ordem crescente de taxa de crescimento, é:

A) f2 – f1 – f3 – f4.
B) f3 – f2 – f4 – f1.
C) f1 – f4 – f3 – f2.
D) f1 – f4 – f2 – f3.
E) f4 – f3 – f1 – f2.

[Poscomp 2023]

QUESTÃO 23 – Considere o seguinte trecho de código:

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
        // instruções O(1)
    }
}
```

Qual das seguintes afirmações é verdadeira sobre a complexidade assintótica desse trecho de código?

A) A complexidade é O(n) se m for uma constante, e O(m) se n for uma constante.
B) A complexidade é O(n log m) se m for uma constante, e O(m log n) se n for uma constante.
C) A complexidade é O(n + m) se n e m forem do mesmo tamanho.
D) A complexidade é O(1) em todos os casos.
E) A complexidade é O(nm) em todos os casos.

Exercício

- Qual a complexidade do algoritmo abaixo:

Input: arrays A and B of n integers each.
Output: Whether or not there is an integer t contained in both A and B.

```
for i := 1 to n do
    for j := 1 to n do
        if A[i] = B[j] then
            return TRUE
return FALSE
```

Exercício

- Qual a complexidade do algoritmo abaixo:

Input: array A of n integers.
Output: Whether or not A contains an integer more than once.

```
for i := 1 to n do
    for j := i + 1 to n do
        if A[i] = A[j] then
            return TRUE
return FALSE
```

Exercicio

- [ZIVIANI] Qual algoritmo você prefere: um algoritmo que requer n^5 passos ou um que requer 2^n?

Exercício

Indique se as afirmativas a seguir são verdadeiras ou falsas e justifique a sua reposta:

a) $2^{n+1} = O(2^n)$

b) $2^{2n} = O(2^n)$

c) $7n^2 = O(n)$

d) $5n^2 + 7n = \Theta(n^2)$

e) $9n^3 + 3n = \Omega(n)$

Referências

- Thomas Cormen, Charles Leiserson and Ronald Rivest. Introduction to Algorithms. Second Edition. McGraw-Hill, 2007.
- Udi Mamber. Introduction to Algorithms: A Creative Approach. Addison-Wesley, 1989.
- Nivio Zizanni. Projeto de Algoritmos com implementações em Java e C++. Tomson, 2006.
- ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Perarson Prentice Halt, v. 3, 2010.
- Notas de Aula Disciplina Estruturas de Dados II: Prof. Dr. Geraldo Braz. DEINF/UFMA
- Notas de Aula Disciplina Algoritmos e Estruturas de Dados II: Prof. Dr. Ítalo Cunha. Departamento de Ciência da Computação. UFMG.
- Notas de Aula Disciplina Análise e Projeto de Algoritmos. Prof. Tiago A. E. Ferreira. Departamento de Estatística e Informática da Universidade Federal Rural de Pernambuco