Análise de Complexidade

Prof. Lilian Berton

São José dos Campos, 2018

Introdução

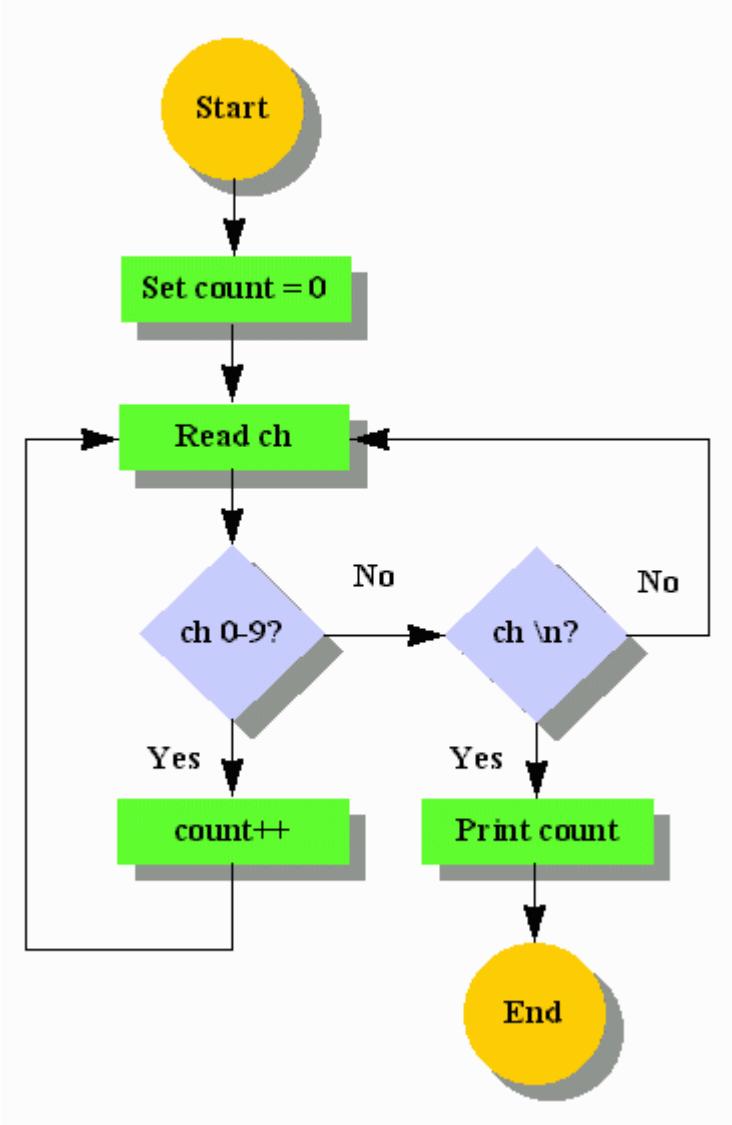
- Quando vamos desenvolver um programa, é necessário e importante estudar as opções de algoritmos a serem utilizados, considerando os aspectos de **tempo de execução e espaço ocupado.**

Rodar programa em um cluster X no celular

Esperar 1hr X 1s para o programa executar

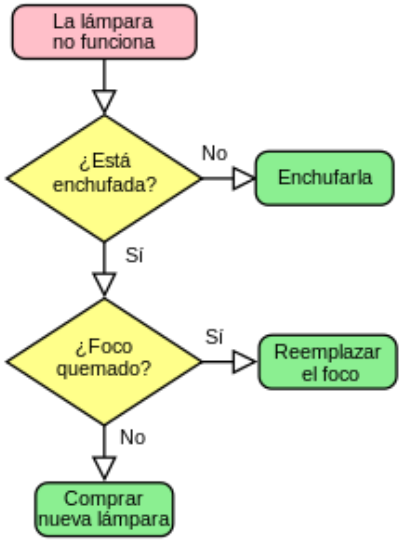
Introdução

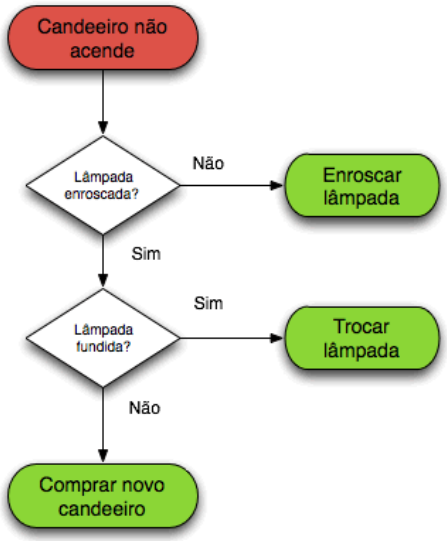
- Análise de um algoritmo em particular.
  - análise do número de vezes que cada parte do algoritmo deve ser executada,
  - estudo da quantidade de memória necessária.



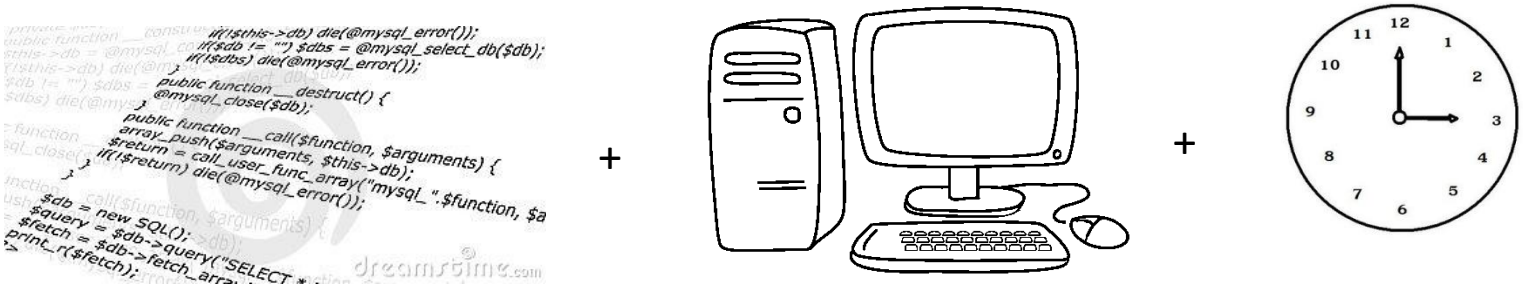
Introdução

- 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.





Medida do custo pela execução do programa



- Os resultados são dependentes do compilador (pode favorecer algumas construções em detrimento de outras) e do hardware;
- Quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.
- 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.

Medida do custo por meio de um modelo matemático

```
procedure Selecao (var A: Vetor; var n: Indice);
var i, j, Min: Indice;
    x        : Item;
for i := 1 to n - 1 do
  begin
  Min := i;
  for j := i + 1 to n do
    if A[j].Chave < A[Min].Chave
    then Min := j;
  x := A[Min]; A[Min] := A[i]; A[i] := x;
```

Nº comparações = $n^2/2 - n/2$

Nº trocas = 3(n-1)

- Usa um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- 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.

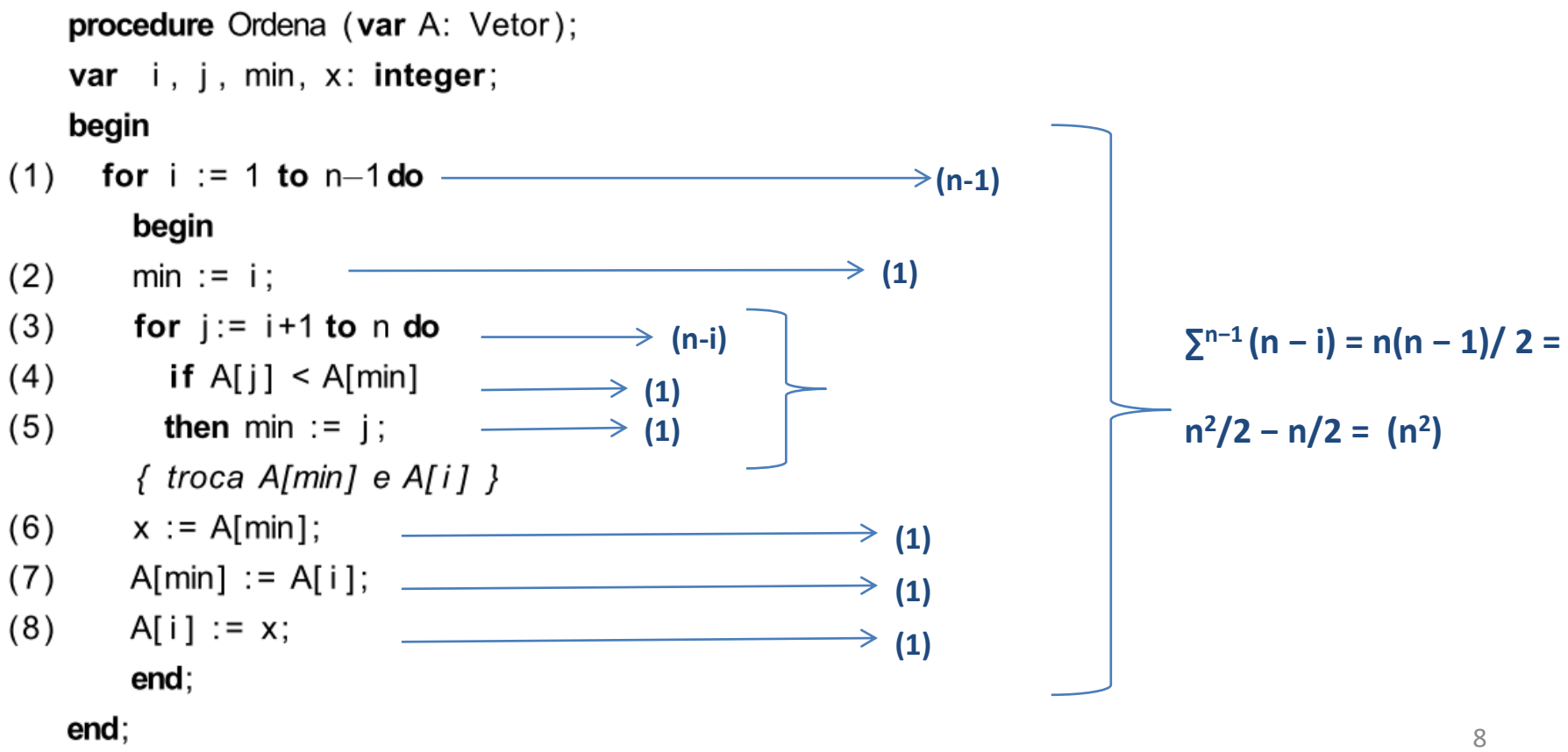
- Função de **complexidade de tempo**: f(n) mede o tempo necessário para executar um algoritmo em um problema de tamanho n.

- Função de **complexidade de espaço**: f(n) mede a memória necessária para executar um algoritmo em um problema de tamanho n.

- A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Função de **complexidade de tempo**

- Algoritmo para ordenar os n elementos de um conjunto A em ordem ascendente.



Análise de tempo

- **Comando de atribuição, de leitura ou de escrita**: O(1).
- **Sequência de comandos**: determinado pelo maior tempo de execução de qualquer comando da sequência.
- **Comando de decisão**: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é O(1).
- **Anel**: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente O(1)), multiplicado pelo número de iterações.
- **Procedimentos não recursivos**: cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos. Avalia-se então os que chamam os já avaliados (utilizando os tempos desses). O processo é repetido até chegar no programa principal.
- **Procedimentos recursivos**: associada uma função de complexidade f(n) desconhecida, onde n mede o tamanho dos argumentos.

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.
- **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.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis.

Exemplo: pesquisa sequencial

- Considere o problema de acessar os registros de um arquivo. Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo. Dada uma chave qualquer, localize o registro que contenha esta chave.

```
1 int pesquisaSequencial(int v[], int tam, int x){
2     int i;
3     for (i=0; i<tam; i++){
4         if ( v[i] == x ){
5             return i;
6         }
7     }
8     return -1;
9 }
```

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: pesquisa sequencial

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.

- Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o i-ésimo registro são necessárias i comparações, então

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

- 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

$$\mathbf{p_i = 1/n,\ 1 \leq i \leq n}$$

- Neste caso

$$\mathbf{f(n) = 1/n\ (1+2+3+ \cdots +n) = 1/n\ (n(n+1)/2) = (n+1)/2}$$

- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

Exemplo: encontrar maior e menor elemento

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros.

```
1  void maxMin1(int v[], int tam, int *max, int *min){
2      int i;
3      *max = *min = v[0];
4      for (i=1; i<tam; i++){
5          if ( v[i] > *max ){
6              *max = v[i];
7          }
8          if ( v[i] < *min ){
9              *min = v[i];
10         }
11     }
12 }
```

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.

Exemplo: encontrar maior e menor elemento

- MaxMin1 pode ser facilmente melhorado: a comparação v[i] < Min só é necessária quando a comparação v[i] > Max dá falso.

```
1  void maxMin1(int v[], int tam, int *max, int *min){
2      int i;
3      *max = *min = v[0];
4      for (i=1; i<tam; i++){
5          if ( v[i] > *max ){
6              *max = v[i];
7          } else if ( v[i] < *min ){
8              *min = v[i];
9          }
10     }
11 }
```

- 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 (v[i] é maior do que Max a metade das vezes. Logo f(n) = n − 1 + (n −1)/2 = 3n/2 − 3/2, para n > 0)

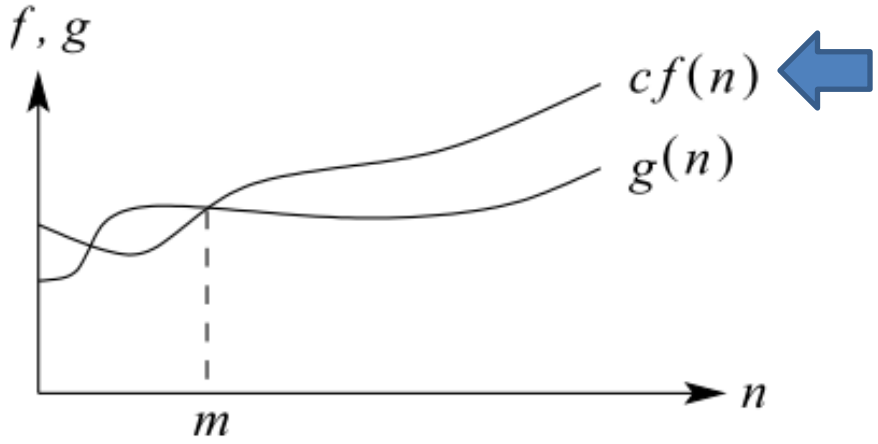
Comportamento assintótico de funções

- 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 assintótico** das funções de custo (comportamento de suas funções de custo para valores grandes de n)
  - O comportamento assintótico de f(n) representa o limite do comportamento do custo quando n cresce.

Dominação assintótica

- **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

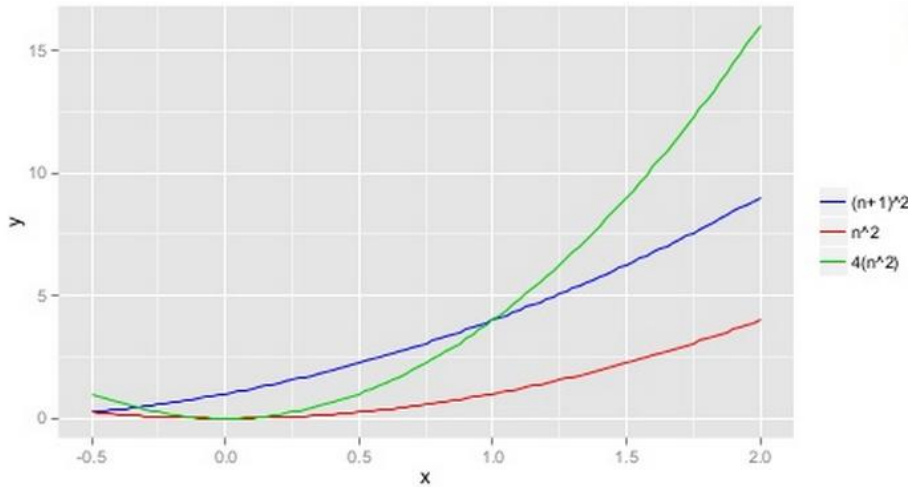
- Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$
- As funções g(n) e f(n) dominam assintoticamente uma a outra, desde que

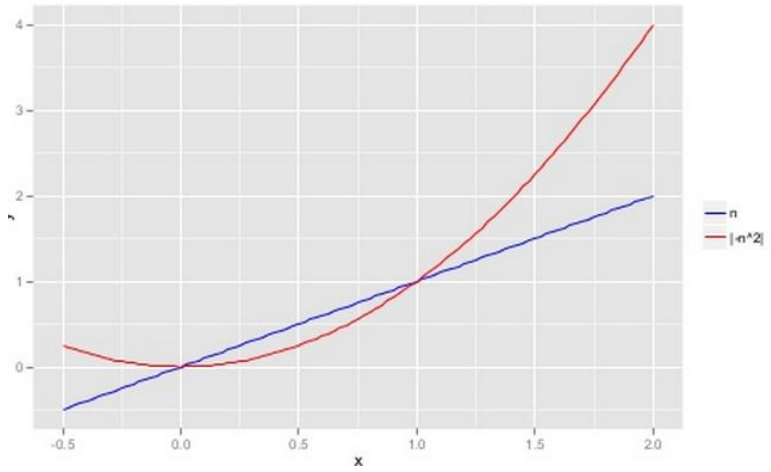
$|(n + 1)^2| \leq 4\ |n^2|$ para $n \geq 1$

e $|n^2| \leq |(n + 1)^2|$ para $n \geq 0$

- Sejam $g(n) = -n^2$ e $f(n) = n$
- A função g(n) domina assintoticamente a f(n), desde que

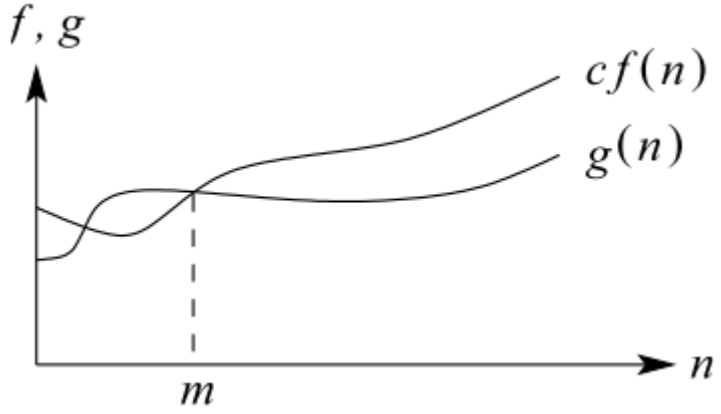
$|n| \leq 1\ |-n^2|$ para $n \geq 1$





Notação O

- **Definição**: Uma função g(n) é O(f(n)) se existem duas constantes positivas c e m tais que $g(n) \leq cf(n)$, para todo $n \geq m$.



- 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 $n \geq m$, $T(n) \leq cn^2$.

Exemplo notação O

- Exemplo: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.
- Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.

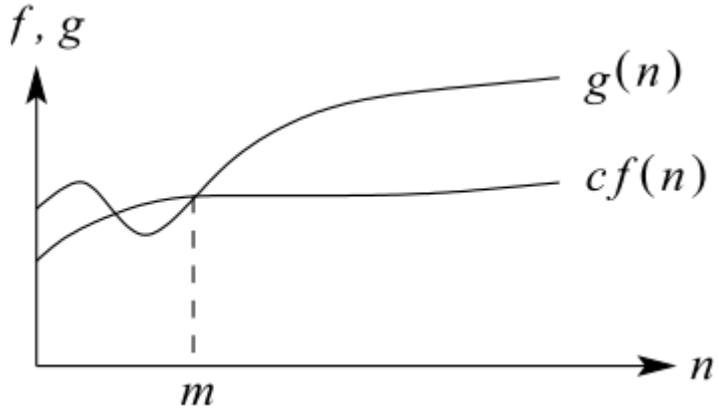
- Exemplo: $g(n) = (n + 1)^2$.
- Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$.
- Pois $(n + 1)^2 \leq 4n^2$ para $n \geq 1$.

Operações com a Notação O

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

Notação Ω

- **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.

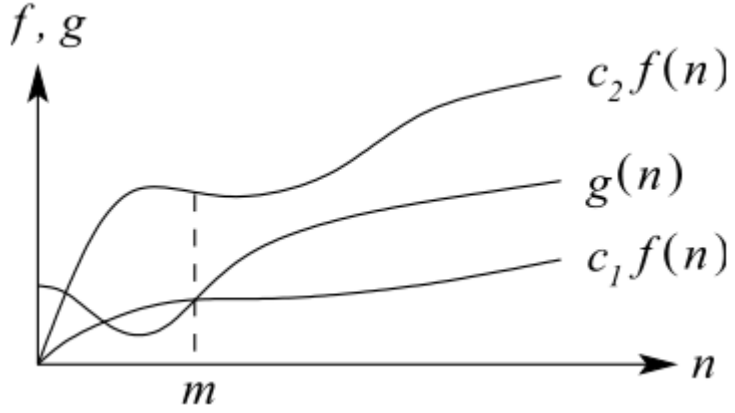


- Exemplo: Para mostrar que g(n) = $3n^3 + 2n^2$ é Ω(n^3) basta fazer c = 1, e então $3n^3 + 2n^2 \geq n^3$ para n ≥ 0.

- 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 Ω(n^2), bastando considerar c = 1/10 e n = 0, 2, 4, 6, . . .

Notação Θ

- **Definição**: Uma função g(n) é Θ(f(n)) se existirem constantes positivas c_1, c_2 e m tais que $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, para todo $n \geq m$.



- Exemplo: para mostrar que $g(n) = n^2/3 - 2n$ é $\Theta(n^2)$, basta escolher $c_1 = 1/21$, $c_2 = 1/3$ e $m = 7$.

Classes de programas

- f(n) = O(1)
  - Um algoritmo de complexidade O(1) é dito ter complexidade constante.
  - Independe do tamanho de n. Instruções são executadas um número fixo de vezes.
  - Ex: inserir um registro.

- f(n) = log(n)
  - Um algoritmo de complexidade log(n) é dito ter complexidade logarítmica.
  - Ocorre em problemas que dividem o problema em problemas menores.
  - Ex: busca binária.

- f(n) = O(n)
  - Um algoritmo de complexidade O(n) é dito ter complexidade linear.
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - Cada vez que n dobra de tamanho, o tempo de execução dobra.
  - Ex: pesquisa sequencial

- $f(n) = O(n \log n)$.
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois.
  - Quando n é 1 milhão, $n \log_2 n$ é cerca de 20 milhões.
  - Ex: Quicksort.

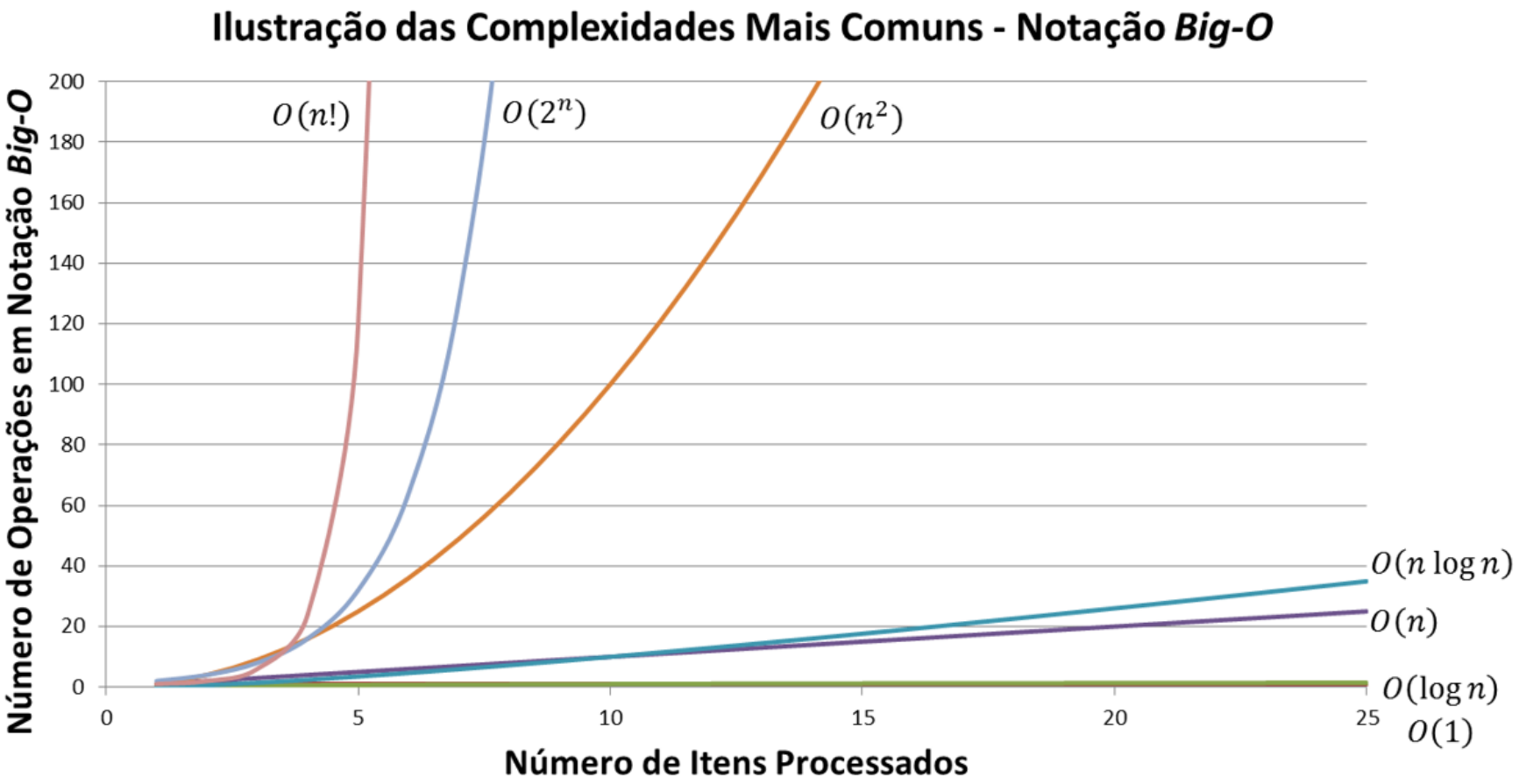
- $f(n) = O(n^2)$
  - Um algoritmo de complexidade $O(n^2)$ é dito ter complexidade quadrática.
  - Ocorre quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
  - Quando n é mil, o número de operações é da ordem de 1 milhão (multiplica por 4).
  - Ex: Seleção, pesquisa em matriz.

- $f(n) = O(n^3)$
  - Um algoritmo de complexidade $O(n^3)$ é dito ter complexidade cúbica.
  - Úteis apenas para resolver pequenos problemas.
  - Quando n é 100, o número de operações é da ordem de 1 milhão (multiplica por 8).
  - Ex: multiplicação de matrizes.

- $f(n) = O(2^n)$
  - Um algoritmo de complexidade $O(2^n)$ é dito ter complexidade exponencial.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa força bruta para resolvê-los.
  - Quando n é 20, o tempo de execução é cerca de 1 milhão (eleva ao quadrado).
  - Ex: problema do caixeiro viajante com programação dinâmica

- $f(n) = O(n!)$
  - Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa força bruta para na solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
  - Ex: problema do caixeiro viajante

Classes de programas



Comparações de funções de complexidade

Função de custo	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	0,0.35 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 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.

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Fonte: Ziviani

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?
  - 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).

Procedimento recursivo

```
    Pesquisa(n);
(1) if n ≤ 1                                              → O(1)
(2) then 'inspecione elemento' e termine                  → O(1)
    else begin
(3)      para cada um dos n elementos 'inspecione elemento'; → O(n)
(4)      Pesquisa(n/3);                                   →
         end;
```

Usa-se uma equação de recorrência para determinar o nº de chamadas recursivas.

- T(n) = n + T(n/3), T(1) = 1 (para n = 1 fazemos uma inspeção)

- T(3) = T(3/3) + 3 = 4
- T(9) = T(9/3) + 9 = 13
- T(27) = T(27/3) + 9 = 13 e assim por diante.

- Para calcular o valor da função seguindo a definição são necessários k − 1 passos para computar o valor de $T(3^k)$.

Resolução equação de recorrência

- Sustitui-se os termos T(k), k < n, até que todos os termos T(k), k > 1, tenham sido substituídos por fórmulas contendo apenas T(1).

$$
\begin{aligned}
T(n) &= n + T(n/3) \\
T(n/3) &= n/3 + T(n/3/3) \\
T(n/3/3) &= n/3/3 + T(n/3/3/3) \\
&\vdots \quad\quad \vdots \\
T(n/3/3\cdots/3) &= n/3/3\cdots/3 + T(n/3\cdots/3)
\end{aligned}
$$

- Adicionando lado a lado, temos
  $T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots + (n/3/3 \cdots /3)$

que representa a soma de uma série geométrica de razão 1/3, multiplicada por n, e adicionada de $T(n/3/3 \cdots /3)$, que é menor ou igual a 1.

Resolução equação de recorrência

- $T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots + (n/3/3 \cdots /3)$

- Se desprezarmos o termo $T(n/3/3 \cdots /3)$, quando k tende para infinito, então

$$T(n) = n \sum_{i=0}^{\infty} (1/3)^i = n \left(\frac{1}{1-\frac{1}{3}} \right) = \frac{3n}{2}.$$

- Logo, o programa do exemplo é O(n).

Você se torna o que você estuda.

Robert Kiyosaki