

Análise de Algoritmos

Prof.: Maiquel de Brito
maiquel.b@ufsc.br

BLU3202 - Algoritmos e Estruturas de Dados
Engenharia de Controle e Automação
Departamento de Engenharias
UFSC Blumenau

Table of contents

1. Introdução
2. Análise assintótica

Introdução

Diferentes algoritmos podem resolver o mesmo problema

ex.: algoritmos de ordenação, pesquisa etc

Como medir se um algoritmo é eficiente?

Tempo de execução, consumo de memória...

dependem da máquina que os executa

Solução: analisar a quantidade de instruções executadas
em função do **tamanho da entrada**

Tamanho da entrada

Tamanho de entrada fixo:

```
1 void dobro(int n){  
2     return n*2;  
3 }
```

Tamanho de entrada variável:

```
1 void dobro(int v[], int n){  
2     for(int i=0;i<n;i++){  
3         printf("%d",v[i]*2);  
4     }  
5 }
```

Desempenho em função do tamanho da entrada (n)

```
1 void teste(int v[], int n){  
2     for(int i=0;i<n;i++){. . . . . c1 n  
3     printf("%d - ",v[i]); . . . . . c2 n  
4 }  
5 }  
6  
7 void main(){  
8     int vetor[] = {1,2,3,4,5};  
9     int tamanho = sizeof(vetor)/sizeof(int);  
10    teste(vetor,tamanho);  
11 }
```

Complexidade da função teste

$$\begin{aligned}C(n) &= c_1 \times n + c_2 \times n \\ &= c_1 n + c_2 n \\ &= (c_1 + c_2)n\end{aligned}$$

Desempenho em função do tamanho da entrada (n)

1	<code>void teste(int v[], int n)</code>	custo	vezes
2	<code>for(int i=0;i<n;i++){</code>	c_1	n
3	<code>for(int j=0;j<n;j++){</code>	c_2	$n \times n$
4	<code>printf("%d - ", v[i]+j);</code>	c_3	$n \times n$
5	<code>}</code>		
6	<code>}</code>		
7	<code>}</code>		
8			
9	<code>void main(){</code>		
10	<code>int vetor[] = {1,2,3,4,5};</code>		
11	<code>int tamanho = sizeof(vetor)/sizeof(int);</code>		
12	<code>teste(vetor, tamanho);</code>		
13	<code>}</code>		

$$\begin{aligned}C(n) &= c_1 \times n + c_2 \times n \times n + c_3 \times n \times n \\&= c_1 n + c_2 n^2 + c_3 n^2 \\&= (c_2 + c_3) n^2 + c_1 n\end{aligned}$$

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?¹

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

¹A cláusula `return` (linhas 4 e 8) provoca o encerramento da execução do algoritmo.

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

1	<code>int primeiro_par(int v[], int n){</code>	custo	vezes
2	<code>for(int i=0;i<n;i++){</code>	c_1	x
3	<code>if(v[i]%2==0){</code>	c_2	x
4	<code>return i;</code>	c_3	1
5	<code>}</code>		
6	<code>}</code>		
7	<code>return -1;</code>	c_4	1
8	<code>}</code>		

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

1	<code>int primeiro_par(int v[], int n){</code>	custo	vezes
2	<code> for(int i=0; i<n; i++){</code>	c_1	x
3	<code> if(v[i]%2==0){</code>	c_2	x
4	<code> return i;</code>	c_3	1
5	<code> }</code>		
6	<code> }</code>		
7	<code> return -1;</code>	c_4	1
8	<code>}</code>		

x – depende da posição do primeiro número par dentro do vetor

melhor caso: menor tempo de execução sobre todas as possíveis entradas de tamanho n

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

1	<code>int primeiro_par(int v[], int n){</code>	custo	vezes
2	<code> for(int i=0; i<n; i++){</code>	c_1	x
3	<code> if(v[i]%2==0){</code>	c_2	x
4	<code> return i;</code>	c_3	1
5	<code> }</code>		
6	<code> }</code>		
7	<code> return -1;</code>	c_4	1
8	<code>}</code>		

x – depende da posição do primeiro número par dentro do vetor

melhor caso: menor tempo de execução sobre todas as possíveis entradas de tamanho n

Na função `primeiro_par`, o melhor caso é quando há um número par na primeira posição do vetor. Ex.: $v = \{2, 3, 4, 5, 6, 7, 8\}$

Nesse caso, $x = 1$ e o custo $C(n) = c_1 \times 1 + c_2 \times 1 + c_3 \times 1 = 3$

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

		custo	vezes
1	<code>int primeiro_par(int v[], int n){</code>		
2	<code> for(int i=0; i<n; i++){</code>	c_1	\times
3	<code> if(v[i]%2==0){</code>	c_2	\times
4	<code> return i;</code>	c_3	1
5	<code> }</code>		
6	<code> }</code>		
7	<code> return -1;</code>	c_4	1
8	<code>}</code>		

↪ Assume-se que cada instrução i têm custo $c_i = 1$.

↪ Assim, pode-se ao menos saber qual a quantidade de instruções executadas.

melhor caso:²

$$\begin{aligned} C(n) &= c_1 \times 1 & + c_2 \times 1 & + c_3 \times 1 \\ &= 1 \times 1 & + 1 \times 1 & + 1 \times 1 = 3 \end{aligned}$$

² No melhor caso, a instrução c_4 nunca é executada e, por isso, não é incluída no cálculo

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

1	<code>int primeiro_par(int v[], int*n){</code>	custo	vezes
2	<code>for(int i=0;i<n;i++){</code>	C_1	x
3	<code>if(v[i]%2==0){</code>	C_2	x
4	<code>return i;</code>	C_3	1
5	<code>}</code>		
6	<code>}</code>		
7	<code>return -1;</code>	C_4	1
8	<code>}</code>		

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

		custo	vezes
1	<code>int primeiro_par(int v[], int*n){</code>		
2	<code>for(int i=0;i<n;i++){</code>	c_1	x
3	<code>if(v[i]%2==0){</code>	c_2	x
4	<code>return i;</code>	c_3	1
5	<code>}</code>		
6	<code>}</code>		
7	<code>return -1;</code>	c_4	1
8	<code>}</code>		

x – depende da posição do primeiro número par dentro do vetor

pior caso: maior tempo de execução sobre todas as possíveis entradas de tamanho n

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

		custo	vezes
1	<code>int primeiro_par(int v[], int*n){</code>		
2	<code> for(int i=0;i<n;i++){</code>	c_1	x
3	<code> if(v[i]%2==0){</code>	c_2	x
4	<code> return i;</code>	c_3	1
5	<code> }</code>		
6	<code> }</code>		
7	<code> return -1;</code>	c_4	1
8	<code>}</code>		

x – depende da posição do primeiro número par dentro do vetor

pior caso: maior tempo de execução sobre todas as possíveis entradas de tamanho n

Na função `primeiro_par`, é (i) quando o primeiro número par está na última posição ou (ii) quando não há número par no vetor. Ex.: $v=\{3,5,7,9,11\}$

Nesse caso, $x = n$ e o custo $C(n) = 2n + 1$

$$\begin{aligned}C(n) &= c_1 \times n + c_2 \times n + c_4 \times 1 \\&= 1 \times n + 1 \times n + 1 \times 1 \\&= 2n + 1\end{aligned}$$

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

	custo	vezes
1 <code>int primeiro_par(int v[], int *n){</code>		
2 <code> for(int i=0; i<n; i++){</code>	c_1	x
3 <code> if(v[i]%2==0){</code>	c_2	x
4 <code> return i;</code>	c_3	1
5 <code> }</code>		
6 <code> }</code>		
7 <code> return -1;</code>	c_4	1
8 <code>}</code>		

A função `primeiro_par` tem custo, no pior caso, de $C(n) = 2n + 1$.

Considerando $C(n)$ a quantidade de instruções executadas em função do tamanho da entrada, qual será essa quantidade para uma entrada de tamanho

- $n = 1000$? $\rightsquigarrow C(1000) = 2 * 1000 + 1 = 2001$ instruções
- $n = 20000$? $\rightsquigarrow C(20000) = 2 * 20000 + 1 = 40001$ instruções

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

	custo	vezes
1 <code>int primeiro_par(int v[], int *n){</code>		
2 <code>for(int i=0; i<n; i++){</code>	C_1	x
3 <code>if(v[i]%2==0){</code>	C_2	x
4 <code>return i;</code>	C_3	1
5 <code>}</code>		
6 <code>}</code>		
7 <code>return -1;</code>	C_4	1
8 <code>}</code>		

A função `primeiro_par` tem custo, no pior caso, de $C(n) = 2n + 1$.

Considerando $C(n)$ o tempo de execução em milissegundos, qual será o tempo gasto para executar a função uma entrada de tamanho

- $n = 1000$? $\rightsquigarrow C(1000) = 2 * 1000 + 1 = 2001$ milissegundos
- $n = 20000$? $\rightsquigarrow C(20000) = 2 * 20000 + 1 = 40001$ milissegundos

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

1	<code>int primeiro_par(int v[], int*n){</code>	custo	vezes
2	<code>for(int i=0;i<n;i++){</code>	c_1	x
3	<code>if(v[i]%2==0){</code>	c_2	x
4	<code>return i;</code>	c_3	1
5	<code>}</code>		
6	<code>}</code>		
7	<code>return -1;</code>	c_4	1
8	<code>}</code>		

Desempenho em função do tamanho da entrada (n)

Qual é o desempenho da função `primeiro_par`?

	custo	vezes
1 <code>int primeiro_par(int v[], int*n){</code>		
2 <code>for(int i=0;i<n;i++){</code>	c_1	x
3 <code>if(v[i]%2==0){</code>	c_2	x
4 <code>return i;</code>	c_3	1
5 <code>}</code>		
6 <code>}</code>		
7 <code>return -1;</code>	c_4	1
8 <code>}</code>		

x – depende da posição do primeiro número par dentro do vetor

caso médio: média dos tempos de execução de todas as entradas de tamanho n .

Requer uma distribuição de probabilidades sobre a entrada.

Se cada posição do vetor tem $\frac{1}{2}$ chances de ter um número par, então:

$$x = 1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 3 \times \frac{1}{8} + \cdots + n \times \frac{1}{2^n} = \sum_{i=1}^n \frac{i}{2^i}$$

$$C(n) = 2 \left(\sum_{i=1}^n \frac{i}{2^i} \right) + 1$$

Desempenho em função do tamanho da entrada (n)

Instruções podem ter custo diferente de 1.

		custo	vezes
1	<code>int conta_dobro(int v[], int n){</code>		
2	<code>int cont=0, i;</code>	1	1
3	<code>for(i=0;i<n;i++){</code>	1	n
4	<code>cont = cont + procura_dobro(v,n,v[i]);</code>	$3n+2$	n
5	<code>}</code>		
6	<code>return cont;</code>	1	1
7	<code>}</code>		
8			
9	<code>int procura_dobro(int v[], int n, int x){</code>		
10	<code>int cont = 0, i;</code>	1	1
11	<code>for(i=0;i<n&&cont==0;i++){</code>	1	n
12	<code>if(v[i]/x==2)</code>	1	n
13	<code>cont=1;</code>	1	n
14	<code>}</code>		
15	<code>return cont;</code>	1	1
16	<code>}</code>		

Custo da função `conta_dobro`: $3n^2 + 3n + 3$

Análise assintótica

Foco: verificar o crescimento do tempo de execução em função do crescimento do tamanho da entrada

Exemplo: Sejam dois algoritmos a_1 e a_2 cujo desempenho é dado, respectivamente por $C_1(n) = 10^6(n) + 10^4$ e $C_2(n) = n^2 + 1$. Qual deles é mais rápido?

Exemplo: Sejam dois algoritmos a_1 e a_2 cujo desempenho é dado, respectivamente por $C_1(n) = 10^6 n + 10^4$ e $C_2(n) = n^2 + 1$. Qual deles é mais eficiente?

n	$C_1(n) = 10^6 n + 10^4$	$C_2(n) = n^2 + 1$
1	1.010.000	2
2	2.010.000	5
\vdots	\vdots	\vdots
1.000.000	$1,00000001 \times 10^{12}$	1×10^{12}
1.000.001	$1,00000101 \times 10^{12}$	$1,00000200 \times 10^{12}$
1.000.002	$1,00000201 \times 10^{12}$	$1,00000400 \times 10^{12}$

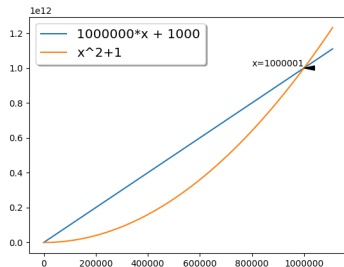
Análise assintótica

Exemplo: Sejam dois algoritmos a_1 e a_2 cujo desempenho é dado, respectivamente por $C_1(n) = 10^6(n) + 10^4$ e $C_2(n) = n^2 + 1$. Qual deles é mais eficiente?

C_2 é menor para um grande número de n

Mas, a partir de um certo n , C_2 passa a ser maior

Pode-se dizer que $C_2(n) > C_1(n)$ quando $n \rightarrow \infty$



Análise assintótica - notação O (big-O):

Formalmente

Sejam $f(n)$ e $g(n)$ duas funções mapeando números inteiros não negativos em números reais. Diz-se que $f(n) = O(g(n))$ se e somente se existem constantes $c > 0$ e $n_0 \geq 1$ tais que $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$

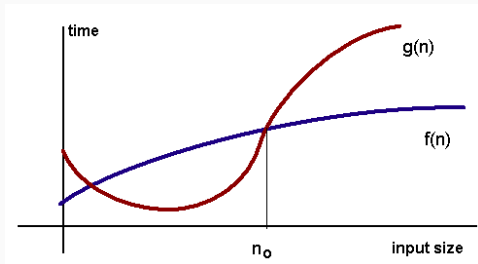
Informalmente

Uma função $f(n)$ é $O(g(n))$ se existe um número n_0 a partir do qual $g(n)$ será sempre maior que $f(n)$.

ou

Uma função $f(n)$ é $O(g(n))$ se $f(n)$ não cresce mais que $g(n)$.

Análise assintótica - notação O (big-O):



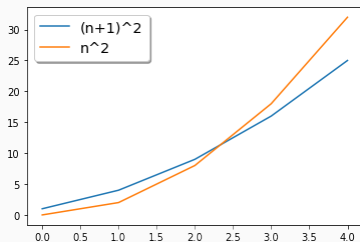
Análise assintótica - notação O (big-O):

Exemplo: Seja $f(n) = (n + 1)^2$ e $g(n) = n^2$.

Pode-se afirmar que $f(n)$ é $O(g(n))$.

Para $c = 2$, tem-se $n_0 \approx 2,42$

n	$f(n) = (n + 1)^2$	$c.g(n) = c.n^2$
1	4	2
2	9	8
3	16	18
4	15	32
\vdots	\vdots	\vdots



Demonstração Aqui

Análise assintótica - notação O (big-O):

Exemplo: Seja $f(n) = 10^6 n^2 + 10^4$ e $g(n) = n^2$.

Pode-se afirmar que $f(n)$ é $O(g(n))$.

para $n \geq 1$ e $c = 1.010.001$.

Pode-se, então afirmar que $f(n)$ é $O(n^2)$

Análise assintótica - notação O (big-O):

Na notação O , despreza-se constantes e termos de mais baixa ordem da função

- O foco está no crescimento da função ($n \rightarrow \infty$)
- Constantes e termos de baixa ordem tornam-se irrelevantes quando $n \rightarrow \infty$

Ex: $f(n) = 3n^2 + 10n + 10$

Para $n = 10$:

- $3n^2 = 73,2\%$ do resultado
- $10n = 24,2\%$ do resultado
- $10 = 2,4\%$ do resultado

Para $n = 100$:

- $3n^2 = 96,7\%$ do resultado
- $10n = 3,2\%$ do resultado
- $10 = 0,1\%$ do resultado

Análise assintótica - notação O (big-O):

Complexidade	Exemplo
$O(\log_n^2)$	Busca binária
$O(n)$	Busca linear
$O(n \log_n^2)$	Quick sort (melhor caso)
$O(n^2)$	Bubble sort
$O(2^n)$	Fibonacci recursivo
$O(n!)$	Gerar todas as permutações em um conjunto

Análise assintótica - notação O (big-O):

n	\log_n	n	$n \log^n$	n^2	n^3	2^n	$n!$
2	1	2	4	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	
32	5	32	160	1024	32768	4294967296	
64	6	64	384	4096	262144	$1,84 \times 10^{19}$	
128	7	128	896	16384	2097152	$3,40 \times 10^{38}$	
256	8	256	2048	65536	16777216	$1,15 \times 10^{77}$	
512	9	512	4608	262144	134217728	$1,34 \times 10^{154}$	
1024	10	1024	10240	1048576	1073741824	$1,79 \times 10^{308}$	

Formalmente

Sejam $f(n)$ e $g(n)$ duas funções mapeando números inteiros não negativos em números reais. Diz-se que $f(n) = \Omega(g(n))$ se e somente se existem constantes $c > 0$ e $n_0 \geq 1$ tais que $f(n) \geq c \cdot g(n)$ para todo $n \geq n_0$

Informalmente

Uma função $f(n)$ é $\Omega(g(n))$ se existe um número n_0 a partir do qual $g(n)$ será sempre *menor* que $f(n)$.

ou

Uma função $f(n)$ é $\Omega(g(n))$ se $g(n)$ não cresce mais que $f(n)$.

Análise assintótica - notação Ω (Omega):

Exemplo: $f(n)$ é $\Omega(g(n))$ porque, para qualquer n à direita de m , $f(n)$ é maior que $c.g(n)$.

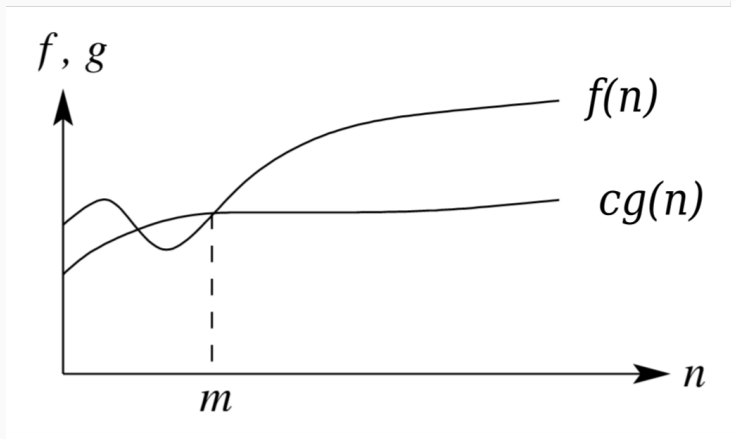


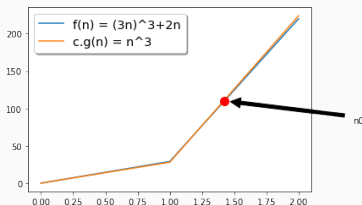
Figure 1: Adaptado de Ziviani, 2010.

Análise assintótica - notação Ω (Omega):

Exemplo: Seja $f(n) = (3n)^3 + 2n$ e $g(n) = n^2$. Pode-se afirmar que $f(n)$ é $\Omega(g(n))$.

Para $c = 28$, tem-se $n_0 \approx 1,42$

n	$f(n) = (3n)^3 + 2n$	$c.g(n) = c.n^2$
0.01	0,020027	0,000001
\vdots	\vdots	\vdots
1.4	76,888	76,832
1.41	78,506967	78,490188
1.42	80,148776	80,172064
\vdots	\vdots	\vdots



Demonstração Aqui

Capítulo 1 de *Ziviani, Nivio. Projeto De Algoritmos Com Implementações Em Pascal E C 3ed.: CENGAGE, 2010. ISBN: 9788522110506.*

Capítulo 6 de *Adamson, Iain T. Data Structures and Algorithms: A First Course. Springer, 1996. ISBN: 978-1-4471-1023-1*

(disponível no acervo eletrônico da BU-UFSC)