

Construção e Análise de Algoritmos

Complexidade de Algoritmos

Professor: Cláudio Soares de Carvalho Neto



Universidade Estadual Vale do Acaraú - UVA
Centro de Ciências Exatas e Tecnologia - CCET
Curso: Ciências da Computação

Sumário

① Algoritmos

Definição

Especificação

Corretude e Eficiência

Complexidade

② Referências

Algoritmos

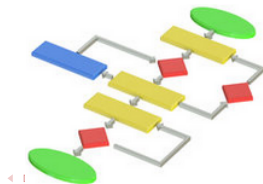
Definição

Informalmente, um *algoritmo* é qualquer procedimento computacional bem definido que recebe um conjunto de valores como entrada e produz um conjunto de valores como saída. (??)

Exemplo: Ordenar uma sequência de números inteiros.

Entrada: Uma sequência $S = \langle a_1, a_2, \dots, a_n \rangle$

Saída: Uma permutação S' de S ,
 $S' = \langle a'_1, a'_2, \dots, a'_n \rangle$
tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.



Especificação

Especificação

Um algoritmo pode ser especificado em linguagem natural, em pseudocódigo, fluxograma etc. O único requisito é que tal especificação forneça uma descrição precisa do procedimento a ser seguido.

Exemplo: Calcular o M.D.C. de dois inteiros positivos.

Dados dois inteiros positivos a e b ;
Enquanto os números forem diferentes,
subtraia o menor do maior.
Ao final desse processo, exiba um dos
números.

```
algoritmo mdc( $a, b$ )  
  enquanto ( $a \neq b$ ) faça  
    se ( $a > b$ ) então  
       $a \leftarrow a - b$   
    senão  
       $b \leftarrow b - a$ ;  
  escreva  $a$ 
```

Corretude e Eficiência

Corretude

Um algoritmo está *correto* se, para toda entrada, produz a saída esperada.

Eficiência

A medida de eficiência de um algoritmo (correto) é a velocidade. Isto é, o tempo que ele leva para produzir o resultado.

Exemplo: Calcular o termo n da Sequência de Fibonacci ($n > 0$).

```
algoritmo fib(n)
  se ( $n < 3$ ) então
    retorne 1;
  senão
    retorne  $fib(n-1) + fib(n-2)$ ;
```

```
algoritmo fib(n)
  alocar vetor  $V[1 \dots n]$ ;
   $V[1] \leftarrow 1$ ;
   $V[2] \leftarrow 1$ ;
  para  $i \leftarrow 3$  até  $n$  faça
     $F[i] \leftarrow F[i-1] + F[i-2]$ ;
  retorne  $F[n]$ 
```

Complexidade

- O estudo de técnicas no desenvolvimento de algoritmos permite prever alguns aspectos da complexidade do algoritmo resultante.
- Por que devemos nos preocupar com a complexidade dos algoritmos?
 - o estudo da complexidade de algoritmos nos permite projetar algoritmos mais eficientes.
 - é possível analisar a complexidade de um algoritmo já construído para que tenhamos uma melhor compreensão da sua eficiência.
- Complexidade de Tempo x Espaço
 - a **complexidade de tempo** é expressa em termos do número de operações que são realizadas pelo algoritmo para um dado tamanho de entrada. Esse número de operações depende da forma como estão os dados na entrada.
 - a **complexidade de espaço** tem a ver com a quantidade de memória utilizada pelo algoritmo para um dado tamanho de entrada.

Complexidade de Tempo

- Tomemos como exemplo o comando $V[i] \leftarrow V[i - 1] + V[i - 2]$.
- Observe que são realizadas pelo menos 7 operações:
 - 2 subtrações
 - 3 cálculos de endereço no vetor V
 - 1 soma
 - 1 atribuição
- Para tornar a tarefa menos exaustiva, podemos nos deter a apenas alguns tipos de operações. Por exemplo, comparações e atribuições. No exemplo acima, contaríamos apenas uma operação.

Exemplo I

Suponha uma máquina executa 10^3 instruções por segundo. O quadro a seguir apresenta o tempo que ela levaria para executar algoritmos de complexidade diferente, para diferentes tamanhos de entrada.

Entrada	Função de Complexidade				
	n	$\log n$	$n \log n$	n^2	2^n
8	0.008 seg	0.003 seg	0.024 seg	0.064 seg	0,256 seg
16	0.016 seg	0.004 seg	0.064 seg	0.256 seg	1,09 min
32	0.032 seg	0.005 seg	0.160 seg	1.024 seg	49,7 dias
64	0.064 seg	0.006 seg	0.384 seg	4.096 seg	5.85×10^6 séculos
128	0.128 seg	0.007 seg	0.896 seg	16.384 seg	1.08×10^{26} séculos

Exemplo II

Agora, consideremos duas máquinas C_1 e C_2 executando algoritmos cujas complexidades são dadas no quadro a seguir. Sabendo-se que C_1 , em um dado tempo t , executa um algoritmo para determinado tamanho de entrada, e que C_2 é 4096 vezes mais rápida que C_1 . Qual o tamanho da maior entrada que C_2 pode resolver no mesmo tempo t ?

Complexidade	Tamanho da maior entrada	
	C_1	C_2
n	a	$4096a$
\log_2^n	b	b^{4096}
n^2	c	$64c$
n^3	d	$16d$
2^n	e	$e + 12$

Exemplo III

- Sejam A_1 e A_2 algoritmos para resolver um mesmo problema, que, para uma entrada de tamanho n , executam, respectivamente, $50n \log_2^n$ e $2n^2$ instruções.
- Sejam C_1 e C_2 computadores que realizam, respectivamente, 10^7 e 10^{10} instruções por segundo.
- Dada uma entrada de tamanho $n = 10^7$, suponha que C_1 use o algoritmo A_1 , e que C_2 use o algoritmo A_2 . Em quanto tempo C_1 e C_2 produzirão o resultado?
 - $\langle C_1, A_1 \rangle : \frac{50 \cdot 10^7 \log_2^{10^7}}{10^7} = 1.166,67 \text{seg} \cong 19.4 \text{ minutos}.$
 - $\langle C_2, A_2 \rangle : \frac{2 \cdot (10^7)^2}{10^{10}} = 20.000 \text{seg} \cong 5 \text{ horas}.$

Cálculo da Função de Complexidade (tempo)

Tomemos como exemplo o algoritmo *BubbleSort*, em que faremos a contagem apenas das operações de atribuição e comparação.

```

**  Algoritmo BubbleSort(A, n)
01  Para  $i \leftarrow n$  até 2 faça
02      Para  $j \leftarrow 1$  até  $i - 1$  faça
03          Se ( $A[j] > A[j + 1]$ ) Então
04              Troca( $A[j], A[j + 1]$ );
  
```

Observação:

A operação troca envolve 3 atribuições.

Linha	Melhor Caso	Pior Caso
01	$2n$	$2n$
02	$(2 + \dots + n) \times 2 = n^2 + n - 2$	$(2 + \dots + n) \times 2 = n^2 + n - 2$
03	$(1 + 2 + \dots + n - 1) = \frac{(n^2 - n)}{2}$	$(1 + 2 + \dots + n - 1) = \frac{(n^2 - n)}{2}$
04	-	$(1 + 2 + \dots + n - 1) \times 3 = \frac{3(n^2 - n)}{2}$
Total	$3n^2/2 + 5n/2 - 2$	$3n^2 + n - 2$

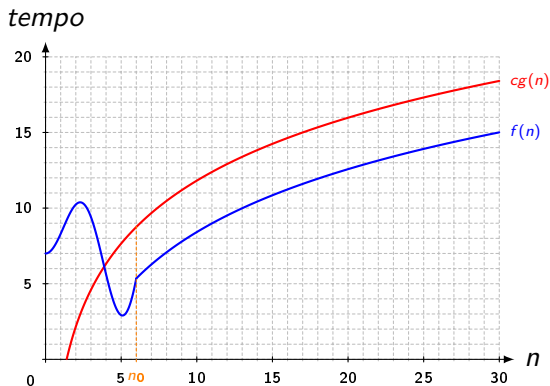
Complexidade do Problema

- A complexidade também pode ser vista como uma propriedade do problema. Nesse caso, a complexidade independe do caminho percorrido na busca da solução.
- Alguns problemas podem ser classificados como “bem comportados” e permitem chegar a bons limites de complexidade.
- Outros problemas parecem ser tão difíceis de serem resolvidos, para instâncias grandes, que parecem se tornar “intratáveis”.

Análise Assintótica

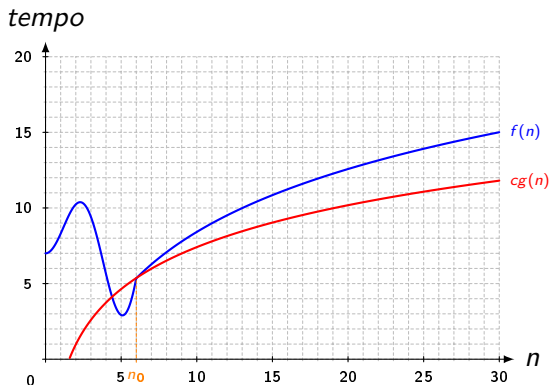
- Ao nos depararmos com funções como $5n^2 + 3n$, $2^n - n^3$, $50n \log n$, é comum pensarmos em valores pequenos para sabermos como elas crescem.
- Em se tratando de análise de algoritmos, concentramos a atenção em valores grandes.
- Na Análise de Algoritmos, chamamos a análise de funções para valores suficientemente grandes de *Análise Assintótica*.
- Nesse contexto, estudaremos as notações O , Ω e Θ .
- Embora não sejam objetos de estudo na disciplina, há também as notações o , ω .

Notação O



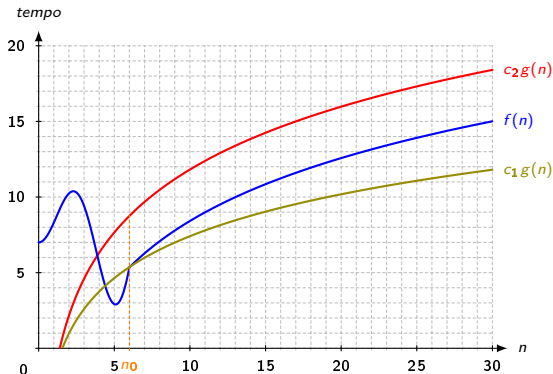
- Uma função $f(n)$ está em $O(g(n))$ se existem constantes positivas c e n_0 , tais que $0 \leq f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$.
- $f(n) = O(g(n))$ indica que $f(n) \in O(g(n))$.
- Limite superior.

Notação Ω



- Uma função $f(n)$ está em $\Omega(g(n))$ se existem constantes positivas c e n_0 , tais que $0 \leq c \cdot g(n) \leq f(n)$ para todo $n \geq n_0$.
- $f(n) = \Omega(g(n))$ indica que $f(n) \in \Omega(g(n))$.
- Limite inferior.

Notação Θ



- Uma função $f(n)$ está em $\Theta(g(n))$ se existem constantes positivas c_1 , c_2 e n_0 , tais que $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ para todo $n \geq n_0$.
- $f(n) = \Theta(g(n))$ indica que $f(n) \in \Theta(g(n))$.
- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.
- Limite restrito.

Propriedades

- Transitividade

- $f(n) = O(g(n))$ e $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$.
- $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.

- Reflexividade

- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.
- $f(n) = \Theta(f(n))$

- Simetria

- $f(n) = \Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$.


- Simetria transposta

- $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$.


Considerações finais


- As notações O , Ω e Θ permitem que nos detenhamos à parte principal da função de complexidade de algoritmos.
- A seguir, temos algumas regras que permitem a simplificação de funções, omitindo termos dominados:
 - constantes multiplicativas podem ser omitidas.
 - n^a domina n^b , se $a > b$.
 - exponenciais dominam polinomiais.
 - polinomiais dominam logaritmos.
- Exemplos:
 - $3n^2 + 5n + 10 = \Theta(n^2)$
 - $2^n + 3^n + 10^6 n^{50} = \Theta(3^n)$
 - $2^{50} = \Theta(1)$

Referências I

 CORMEN, T. H. et al. *Introduction to Algorithms*. 3. ed. Cambridge, Massachusetts: MIT Press, 2009.

 DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. V. *Algorithms*. 1. ed. New York: McGraw-Hill, 2008.

 GOODRICH, M. T.; TAMASSIA, R. *Projeto de Algoritmos - Fundamentos, análises e exemplos da Internet*. 1. ed. Porto Alegre: Bookman, 2004.

 TOSCANI, L. V.; VELOSO, P. A. S. *Complexidade de Algoritmos*. 3. ed. Porto Alegre: Bookman, 2012. v. 13.