Complexidade de Algoritmos

Análise de Algoritmos

Prof. Dr. Osvaldo Luiz de Oliveira

Estas anotações devem ser
complementadas por
apontamentos em aula.

Motivação

Seqüência de Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, ...

Algoritmo 1

Algoritmo Fib (n)

Entrada: n, inteiro, $n \geq 1$.

Saída: retorna o elemento de ordem n da seqüência de Fibonacci.

```
{
    a := 1; aa := 1; f := 1; i := 3;
    enquanto ( i ≤ n )
    {
       aux := f; f := a + aa; aa := a; a := aux;
       i := i + 1
    }

    retornar f
}
```

Algoritmo 2

Algoritmo Fib (n)

Entrada: n, inteiro, $n \geq 1$.

Saída: retorna o elemento de ordem n da seqüência de Fibonacci.

{
    se ($n \leq 2$)
        retornar 1
    senão
        retornar Fib ($n - 1$) + Fib ($n - 2$)
}

Usando uma calculadora, em quanto tempo você calcularia o número de ordem 100 da seqüência de Fibonacci?

Um computador capaz de realizar 10^6 somas por segundo gastaria, executando o:

- Algoritmo 1: alguns milissegundos.
- Algoritmo 2: centenas de milhares de anos.

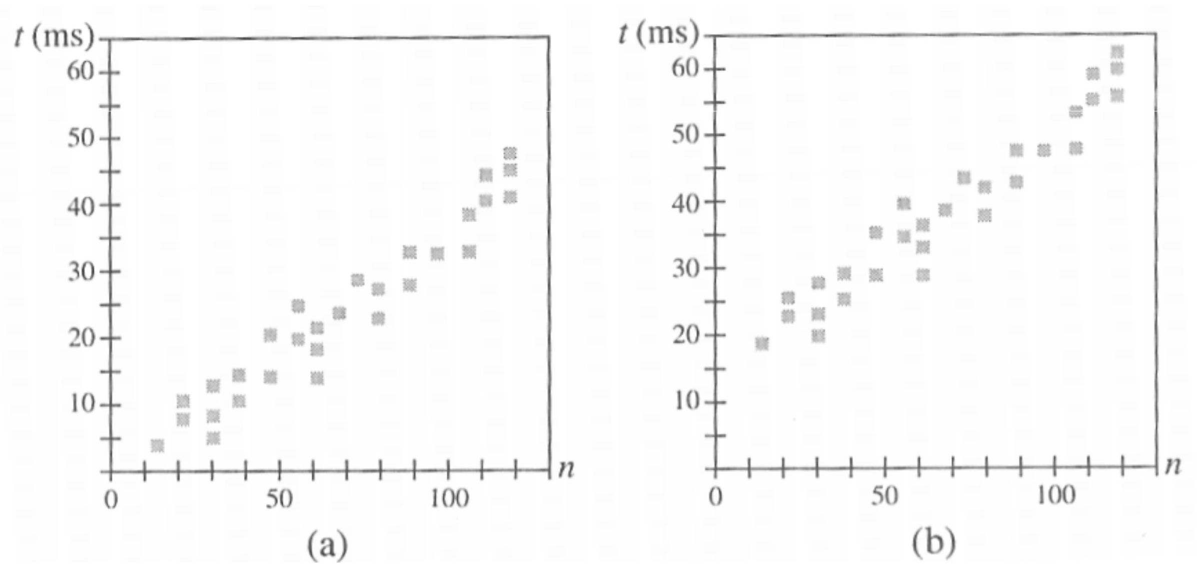
Analisar um algoritmo

- Investigar e definir o comportamento:
  - tempo;
  - de espaço (consumo de memória);
  - outras propriedades.

Metodologias para analisar algoritmos

Experimental

- Mede-se o tempo (*t*) de execução do algoritmo por um computador para diferentes tamanhos de entrada (*n*).



Fonte: GOODRICH, M. T.; TAMASSIA, R. **Algoritm Desin: foondations, analysis and Internet examples**. New York: John Wiley & Sons, 2002.

Que dificuldades você vê nesta abordagem?

Teoria da Complexidade de Algoritmos

- Modelo de computação (abstrato, não se prende a um certo computador ou tecnologia).

- Objetivo: calcular uma função matemática $f(n)$ que descreve o comportamento do algoritmo considerando um modelo de computação (n representa o tamanho da entrada).

A análise conduzirá

Algoritmo Fib (n)
Entrada: n, inteiro, $n \geq 1$.
Saída: retorna o elemento de ordem n da seqüência de Fibonacci.

```
{
    a := 1; aa := 1; f := 1; i := 3;
    enquanto ( i ≤ n )
    {
        aux := f; f := a + aa; aa := a; a := aux;
        i := i + 1
    }

    retornar f
}
```

Tempo de execução do Algoritmo 1:

$T(n) = O(n)$.

A análise conduzirá

Algoritmo Fib (n)
Entrada: n, inteiro, $n \geq 1$.
Saída: retorna o elemento de ordem n da seqüência de Fibonacci.
{
    se ($n \leq 2$)
        retornar 1
    senão
        retornar Fib ($n - 1$) + Fib ($n - 2$)
}

Tempo de execução do Algoritmo 2:

$T(n) = O\ (2^n)$.

Quantas vezes um laço é executado?

Quantas vezes o comando “C” será executado?

```
i := 1;
enquanto ( i ≤ 10 )
{
     C;
      i := i + 1
}
```

Quantas vezes o comando “C” será executado?

i := 4;
enquanto (i ≤ n)
{
    C;
    i := i + 2
}

Progressão Aritmética (PA)

- Seja a sequência $a_1, a_2, a_3, ..., a_k$, onde $a_i = a_{i-1} + r$, para $2 \leq i \leq k$ e r constante.

- Exemplo 1:
  1, 3, 5, 7, 9:
  sequência em que
  - $a_1 = 1$, $a_2 = 3$, $a_3 = 5$, ..., $a_k = a_5 = 9$.
  - $k = 5$.
  - $r = 2$.

- Exemplo 2:
  1, 2, 3, ..., n:
  sequência em que
  - $a_1 = 1$, $a_2 = 2$, $a_3 = 3$, ..., $a_k = n$.
  - $k = n$.
  - $r = 1$.

- Exemplo 3:
  9, 7, 5, 3, 1 :
  sequência em que
  - $a_1 = 9$, $a_2 = 7$, $a_3 = 5$, ..., $a_k = a_5 = 1$.
  - $k = 5$.
  - $r = -2$.

Fórmulas úteis em PA

- Último termo:

  $a_k = a_1 + (k - 1)\ r.$

  Último

  Primeiro

  Quantidade

  Razão

- Soma dos k termos da PA:

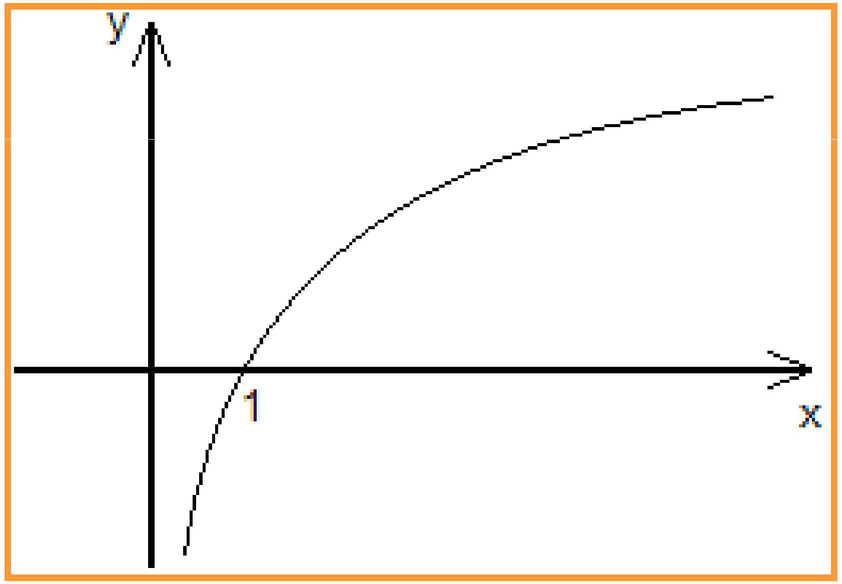
  $S(k) = a_1 + a_2 + a_3 + \ldots + a_k = k\ (a_k + a_1)/2.$

Quantas vezes o comando "C" será executado?

```
i := 1;
enquanto ( i ≤ n )
{
    C;
    i := i * 2
}
```

Logaritmos

$\log_b x = y$ se e somente se $b^y = x$.



Propriedades dos logaritmos

$$\log_c a.b = \log_c a + \log_c b.$$

$$\log_c a / b = \log_c a - \log_c b.$$

$$\log_b a = \frac{1}{\log_a b}.$$

$$\log_a x = \frac{\log_b x}{\log_b a}.$$

$$b^{\log_b x} = x.$$

$$b^{\log_a x} = x^{\log_a b}.$$

Progressão Geométrica (PG)

- Seja a sequência $a_1, a_2, a_3, ..., a_k$, onde $a_i = a_{i-1}\ r$, para $2 \leq i \leq k$ e r constante.

- Exemplo 1:
  1, 2, 4, 8, 16:
  sequência em que
  - $a_1 = 1$, $a_2 = 2$, $a_3 = 4$, ..., $a_k = a_5 = 16$.
  - $k = 5$.
  - $r = 2$.

- Exemplo 2:
  1, 2, 4, 8, ..., 2^n:
  sequência em que
  - $a_1 = 1$, $a_2 = 2$, $a_3 = 3$, ..., $a_k = 2^n$.
  - $k = n + 1$.
  - $r = 2$.

- Exemplo 3:
  16, 8, 4, 2, 1:
  sequência em que
  - $a_1 = 16$, $a_2 = 8$, $a_3 = 4$, ..., $a_k = a_5 = 1$.
  - $k = 5$.
  - $r = 1/2$.

Fórmulas úteis em PG

- Último termo:

  $a_k = a_1\ r^{\,k\,-\,1}$.

  Último — Primeiro — Razão — Quantidade

- Soma dos *k* termos da PA:

  $S(k) = a_1 + a_2 + a_3 + \ldots + a_k = a_1(r^k - 1)/(r - 1)$.

- Se $0 < r < 1$ então a soma de infinitos termos da PG é

  $S = a_1 / (1 - r)$.

Quantas vezes o comando “C” será executado?

```
para i := 1 até n faça
    para j := 1 até m faça
        C
```

Quantas vezes o comando "C" será executado?

para i := 1 **até** *n* **faça**
    para j := i **até** *n* **faça**
        C

Modelo de Computação RAM
(usaremos)

Um autômato no qual:

- instruções são executadas uma após a outra. nenhuma operação ocorre em paralelo.
- um único processador executa as operações.
- o tempo de acesso é uniforme para todas as localizações de memória.

Qual a complexidade de tempo?

Algoritmo Potencia_de_2
Entrada: lê n, inteiro, $n \geq 1$.
Saída: escreve 2^n.

```
{
    ler (n);
    p := 1;
    enquanto ( n > 0 )
    {
        p := 2*p;
        n := n - 1
    }
    escrever p
}
```

Qual a complexidade de tempo?

- Sejam constantes t1, t2, t3, t4, t5 e t6 representando tempos.

Qual a complexidade de tempo?

Algoritmo Potencia_de_2
Entrada: lê n, inteiro, $n \geq 1$.
Saída: escreve 2^n.

{
    ler (n); — t1
    p := 1; — t2
    enquanto ($n > 0$) — t3
    {
        p := 2*p; — t4
        n := n - 1 — t5
    }
    escrever p — t6
}

Complexidade de tempo T(n).

T(n) = t1 + t2 +
n (t4 + t5) +
(n + 1) t3 +
t6

Qual a complexidade de tempo?

- T(n) = t1 + t2 + t6 + *n* t4 + n t5 + *n* t3 + t3.

- T(n) = t1 + t2 + t6 + t3 + *n* (t4 + t5 + t3).

- Sejam constantes c1 = t1 + t2 + t6 + t3 e
  c2 = t4 + t5 + t3.

- T(*n*) = c1 + c2 *n*.

O mesmo algoritmo na linguagem da RAM

ler (n);		READ 1
		LOAD $1
p := 1;		STORE 2
enquanto ($n > 0$)	L:	LOAD 1
		JPZ F
		LOAD 2
		MUL $2
p := 2*p;		STORE 2
		LOAD 1;
		SUB $1
$n := n - 1$		STORE 1
		JP L
escrever (p)	F:	WRITE 2
		HALT

O mesmo algoritmo na linguagem da RAM

```
    READ  1      ; lê da entrada padrão e armazena na posição de memória 1.
    LOAD  $1     ; carrega no registrador A a constante 1.
    STORE  2     ; armazena na posição de memória 2 o valor do registrador A.
L:  LOAD  1      ; carrega no registrador A o valor da posição de memória 1.
    JPZ  F       ; desvia para a instrução de rótulo “F” se o valor do registrador
                 ; A for igual a zero.
    LOAD 2       ; carrega no registrador A o valor da posição de memória 2.
    MUL $2       ; multiplica o valor do registrador A pela constante 2.
    STORE 2
    LOAD 1;
    SUB  $1      ; subtrai o valor do registrador pela constante 1.
    STORE 1
    JP  L        ; desvia para a instrução de rótulo “L”.
F:  WRITE  2     ; escreve na saída padrão o valor da posição de memória 2
    HALT         ; encerra a execução.
```

Qual a complexidade de tempo?

```
      READ  1
      LOAD  $1
      STORE  2
L:    LOAD  1
      JPZ  F
      LOAD 2
      MUL $2
      STORE 2
      LOAD 1;
      SUB  $1
      STORE 1
      JP  L
F:    WRITE  2
      HALT
```

Teste
$n > 0$

Suponha que cada instrução executa em um tempo constante igual a t.

Complexidade de tempo T(n).

$$T(n) = 3\,t + 7\,t\,n + 2\,t\,(n + 1) + 2\,t$$

$$T(n) = 3\,t + 7\,t\,n + 2\,t\,n + 2\,t + 2\,t$$

$$T(n) = 7\,t + 9\,t\,n$$

$T(n) = c1 + c2\, n$,
para constantes $c1 = 7\,t$ e $c2 = 9\,t$.

Resultado importante
(não será provado)

O tempo de execução de cada comando de em uma linguagem de alto nível é proporcional ao tempo de execução de comandos da máquina do modelo RAM.

⇓

Logo podemos trabalhar sobre algoritmos escritos em linguagem de alto nível como se estivéssemos trabalhando sobre a máquina do modelo RAM.

Ignorando fatores constantes

• Nós iremos **ignorar fatores constantes** e nos concentraremos no comportamento assintótico da função que descreve a complexidade do algoritmo.

• Por exemplo, se o algoritmo gasta $100n$ passos para retornar o resultado, então nós diremos que o tempo de execução do algoritmo será descrito por n (i.e., é proporcional a n).

Complexidades

Um algoritmo pode ter um comportamento diferente para entradas de tamanho igual a n.

- Pior caso (ou pessimista);
- Melhor caso (ou otimista);
- Caso médio.

Calcule as complexidades de pior caso, melhor caso e caso médio

Algoritmo Busca (A, n, x)

Entrada: procura o valor igual a x no vetor "A", de "n" elementos.

Saída: retorna o índice do elemento do vetor "A" igual a "x" ou retorna zero, caso tal elemento não exista.

```
{
    i := 1;
    enquanto ( i ≤ n  e  x ≠ A[i] )
        i := i + 1;

    se ( i ≤ n )
        retornar i
    senão
        retornar 0
}
```

Qual a complexidade de espaço?

Algoritmo Busca (A, n, x)

Entrada: procura o valor igual a x no vetor “A”, de “n” elementos.

Saída: retorna o índice do elemento do vetor “A” igual a “x” ou retorna zero, caso tal elemento não exista.

{

  i := 1;

  enquanto ($i \leq n$ e $x \neq A[i]$)

    i := i + 1;

  se ($i \leq n$)

    retornar i

  senão

    retornar 0

}

Outros modelos de computação
(não usaremos)

- PRAM: parallel RAM.

- Modelo de bits: debita 1 para cada operação sobre um bit.

Crescimento de funções

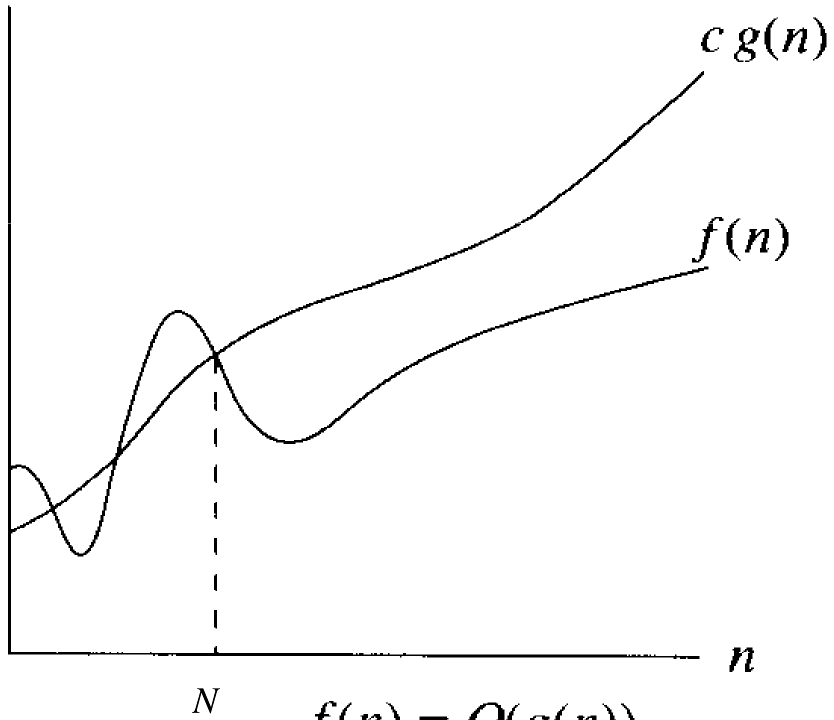
Notação *O - limite superior -* “ $\leq$ ”

Sejam f e g funções.

Definição: $f\,(n)$ é $O\,(\,g\,(n)\,)$ se e somente se existem constantes $c > 0$ e N tais que para todo $n \geq N$ nós temos
$f\,(n) \leq c.g\,(n)$.

Observação: na maioria das vezes escreveremos
$f\,(n) = O\,(\,g\,(n)\,)$ para significar que $f\,(n)$ é $O\,(\,g\,(n)\,)$.

Graficamente



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

Exemplos

- $2n^2 + 4$ é $O(n^2)$, pois $2n^2 + 4 \leq 3n^2$ para $n \geq 2$ ($c = 3$, $N = 2$)

- $2n^2 + 4$ é $O(n^3)$, pois $2n^2 + 4 \leq 1n^3$ para $n \geq 3$ ($c = 1$, $N = 3$)

Mais observações

- A notação O permite ignorar constantes convenientemente. Assim:
  - $O(n) = O(5n + 10)$
  - $O(\log_2 n) = O(\log_3 \mathrm{n}) = O(\log_4 \mathrm{n}) = \ldots$

- $O(1)$ denota uma constante.

- Algumas vezes usamos a notação O como parte de uma expressão:
  - $T(n) = 3n^2 + O(n)$
  - $S(n) = 2n\log_2 n + 5n + O(1)$

Provando que uma função $f(n) = O(g(n))$

Primeira ferramenta:

nós aplicamos a definição, isto é, temos que achar c e N que satisfaçam a definição.

Segunda ferramenta

Teorema 1: para todas as constantes $c > 0$ e $a > 1$ e para todas funções monotonamente crescentes $f(n)$ vale

$$[f(n)]^c = O(a^{f(n)}).$$

Ou seja, uma função exponencial cresce mais rápido ou tanto quanto uma função polinomial. Uma função é monotonamente crescente se somente se para todo $n_2 \geq n_1$ temos que $f(n_2) \geq f(n_1)$.

Exercícios:

1 - Mostre que $n^2 = O(2^n)$.

2 - Mostre que $\log_2 n = O(n)$.

Terceira ferramenta

Teorema 2: se $f(n) = O(s(n))$ e $g(n) = O(r(n))$ então
$f(n) + g(n) = O(s(n) + r(n))$.

Teorema 3: se $f(n) = O(s(n))$ e $g(n) = O(r(n))$ então
$f(n) \,.\, g(n) = O(s(n) \,.\, r(n))$.

Tempo de computação para $n = 1000$

running times	$time_1$ 1000 steps/sec	$time_2$ 2000 steps/sec	$time_3$ 4000 steps/sec	$time_4$ 8000 steps/sec
$\log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Fonte: MANBER, U. **Introduction to Algorithms: A Creative Approach**. Boston: Addison Wesley, 1989.

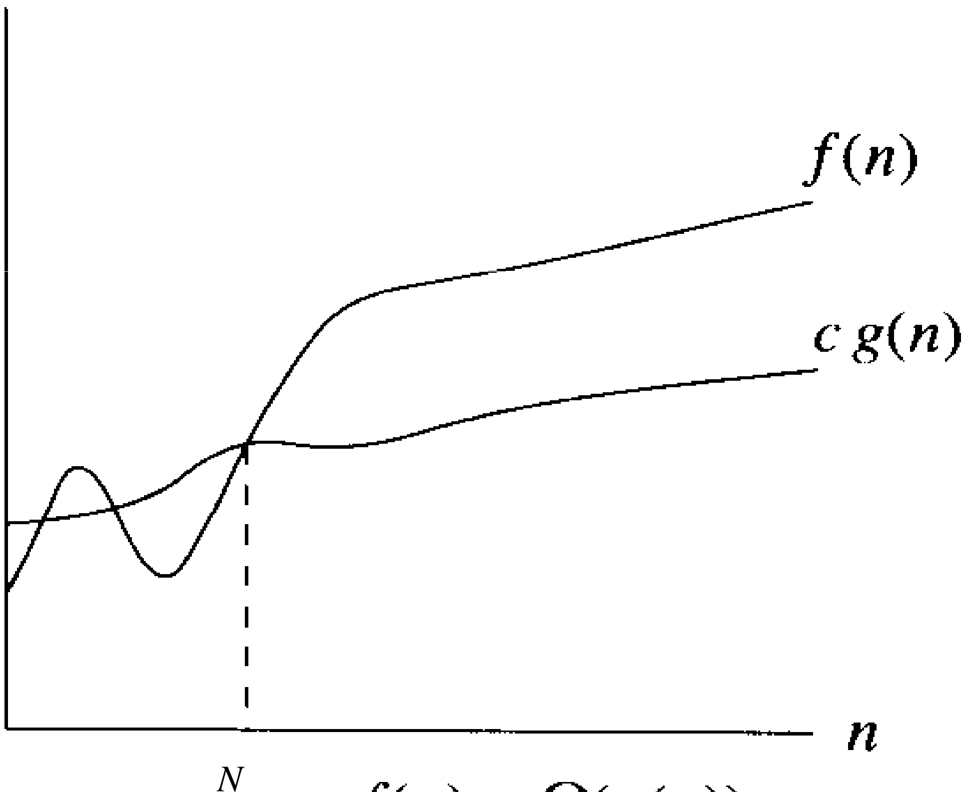
Um algoritmo com tempo de execução exponencial irá gastar um tempo astronômico (bilhões e bilhões de anos) para resolver um problema com o tamanho da entrada $n = 1000$ (mesmo que a base esteja próxima de 1).

Notação Ω - *limite inferior* - “ ≥ ”

Sejam f e g funções.

Definição: $f\,(n)$ é $\Omega\,(\,g\,(n)\,)$ se e somente se existem constantes $c > 0$ e N tais que, para todo $n \geq N$ nós temos $f\,(n) \geq c.g\,(n)$.

Graficamente



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

Exemplos

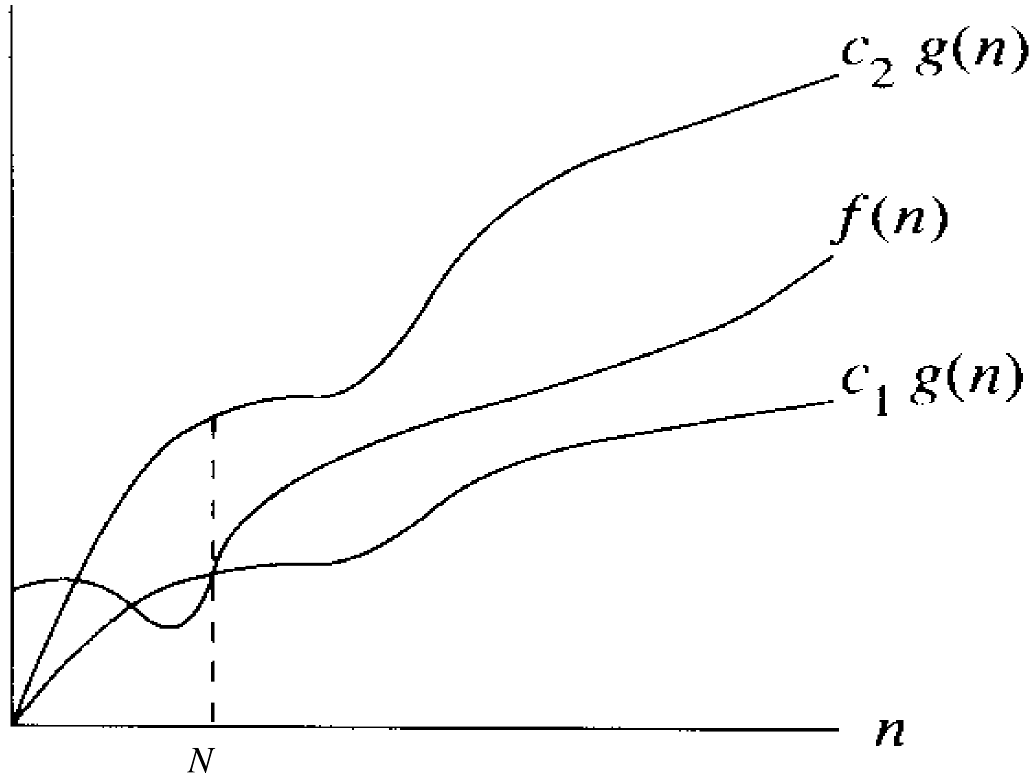
- $2n^2 + 4$ é $\Omega(n^2)$, pois $2n^2 + 4 \geq 1n^2$ para $n \geq 0$ ($c = 1$, $N = 0$)
- n^3 é $\Omega(n^2)$, pois $n^3 \geq 1n^2$ para $n \geq 1$ ($c = 1$, $N = 1$)

Notação Θ - " = "

Sejam f e g funções.

Definição: $f\,(n)$ é $\Theta\,(g\,(n))$ se somente se $f\,(n)$ é $O\,(g\,(n)\,)$ e $f\,(n)$ é $\Omega\,(\,g\,(n)\,)$.

Graficamente



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

Exemplo

$2n^2 + 4$ é $\Theta(n^2)$ pois, como mostramos anteriormente, que:

- $2n^2 + 4$ é $O(n^2)$ e que
- $2n^2 + 4$ é $\Omega(n^2)$.

Exercício:

1 - Mostre que $5n \log n - 100$ é $\Theta(n \log n)$.

Notações o, "$<$", e ω, "$>$"

Sejam f e g funções.

Definição 1: $f(n)$ é $o(g(n))$ se e somente se para todo $c > 0$ vale $f(n) < c.g(n)$ para $n \geq N$ (c e N constantes).

Definição 2: $f(n)$ é $\omega(g(n))$ se e somente se para todo $c > 0$ vale $f(n) > c.g(n)$ para $n \geq N$ (c e N constantes).

Provando que $f(n) = o(g(n))$

Teorema 4: $f(n) = o(g(n))$ se $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Teorema 5: para todas as constantes $c > 0$ e $a > 1$ e para todas funções monotonamente crescentes $f(n)$ vale

$$[f(n)]^c = o(a^{f(n)}).$$

Ou seja, uma função exponencial cresce mais rápido do que uma função polinomial.

Somatórios e outras ferramentas matemáticas úteis

Séries aritméticas:

$1 + 2 + 3 + ... + n = n\ (n + 1)/2$

Genericamente, se $a_k = a_{k\text{-}1} + r$, onde r é uma constante, então

$a_1 + a_2 + a_3 + ... + a_k = n\ (a_k + a_1)/2$.

Séries geométricas:

$1 + 2 + 4 + ... + 2^n = 2^{n+1} - 1$

Genericamente, se $a_k = r\ a_{k\text{-}1}$, onde r é uma constante, então

$a_1 + a_2 + a_3 + ... + a_k = a_1\ (r^{\mathrm{k}} - 1) / (r - 1)$.

Soma dos quadrados:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

Séries harmônicas:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n),$$

Onde $\gamma = 0.577...$ é a constante de Euler.

Logaritmos:

$$\log_b a = x \Leftrightarrow b^x = a.$$

$$\log_c a.b = \log_c a + \log_c b.$$

$$\log_c a / b = \log_c a - \log_c b.$$

$$\log_b a = \frac{1}{\log_a b}.$$

$$\log_a x = \frac{\log_b x}{\log_b a}.$$

$$b^{\log_b x} = x.$$

$$b^{\log_a x} = x^{\log_a b}.$$

Soma de logaritmos:

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 = \Theta(n \log n).$$

Limites de uma soma por meio de integral:

Quando $f(k)$ é monotonamente crescente.

$$\int_{m-1}^{n} f(x)\,dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)\,dx\ .$$

Quando $f(k)$ é monotonamente decrescente.

$$\int_{m}^{n+1} f(x)\,dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)\,dx\ .$$

Regra da cadeia:

$$\int u dv = u.v - \int v du.$$

Aproximação de Stirling:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + O(1/n)\right).$$

Em particular, a aproximação de Stirling implica que

$$\log_2(n!) = \Theta(n \log n).$$