COMPLEXIDADE DE ALGORITMOS

Algoritmos

• Seqüência de instruções necessárias para a resolução de um problema bem formulado

• Permite implementação computacional

COMPLEXIDADE DE ALGORITMOS

• Um algoritmo resolve o problema quando para qualquer entrada produz uma resposta correta

• Mesmo resolvendo um problema, um algoritmo pode não ser aceitável na prática por requerer muito espaço e tempo

• Um problema é considerado INTRATÁVEL, se não existe um algoritmo para ele cuja demanda de recursos computacionais seja razoável.

COMPLEXIDADE DE ALGORITMOS

Questões

- O problema em questão é tratável?
- Existe um algoritmo que demande quantidade razoável de recursos computacionais?
- Quais os custos deste algoritmo?
- Existe um algoritmo melhor?
- Como comparar algoritmos?

COMPLEXIDADE DE ALGORITMOS

Exercício:

$$1! + 2! + 3! + \ldots + n!$$

COMPLEXIDADE DE ALGORITMOS

Estas e outras questões são abordadas em uma área de conhecimento denominada

Análise de complexidade de algoritmos

COMPLEXIDADE DE ALGORITMOS

- O custo final de um algoritmo pode estar relacionado a diversos fatores:
  - Tempo de execução
  - Utilização de memória principal
  - Utilização de disco
  - Consumo de energia, etc...
- Medidas importantes em outros contextos:
  - legibilidade do código
  - custo de implementação
  - portabilidade
  - extensibilidade

COMPLEXIDADE DE ALGORITMOS

Exemplo:

Solução de um sistema de equações lineares
AX = 0

Métodos:

Cramer (determinantes)
Gauss (escalonamento)

COMPLEXIDADE DE ALGORITMOS

Estimativa empírica de tempo, para um processador rodando em 3GHz

n	Cramer	Gauss
2	4 ns	2 ns
3	12 ns	8 ns
4	48 ns	20 ns
5	240ns	40 ns
10	7.3ms	330 ns
20	152 anos	2.7 ms

Em maioria dos casos, complexidade em tempo é preocupação principal

COMPLEXIDADE DE ALGORITMOS

Outro exemplo

- Busca seqüencial
- Dado um vetor de números, verificar se um número chave encontra-se neste vetor

```
char achou = 0;
i = 0;
while (!achou && i<n) {
   achou = (vet[i] == chave);
   i++;
}
if (achou) return(i);
else return(1);
```

COMPLEXIDADE DE ALGORITMOS

Outro exemplo

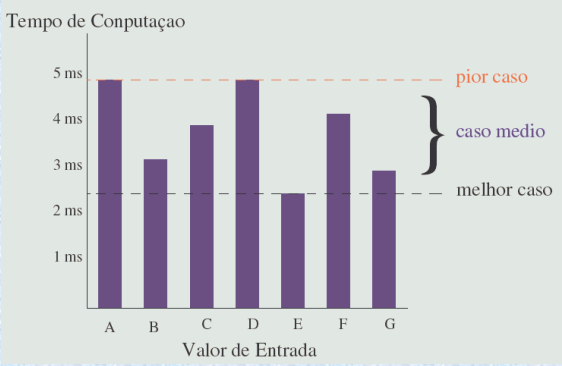
- Busca seqüencial
  - Neste caso não existe um número fixo de operações!
  - Isto vai depender de onde o valor chave é encontrado
  - Por isso, é comum realizar a contagem para:
    - O melhor caso
    - O pior caso
    - E o caso médio

TEMPO DE EXECUÇÃO DE ALGORÍTMOS

- Um algoritmo pode rodar mais rápido para certos conjunto de dados do que para outros.
- Encontrar um *caso médio* pode ser muito difícil, assim os algoritmos são geralmente medidos pela complexidade de tempo do *pior caso*.

TEMPO DE EXECUÇÃO DE ALGORÍTMOS

Além disso, para certas áreas de aplicação (controle de tráfego aérea, cirurgias, etc.), o conhecimento da complexidade do pior caso é crucial.



COMPLEXIDADE DE ALGORITMOS

- Análise de Complexidade de tempo
  - Análise Experimental
  - Análise Teórica

COMPLEXIDADE DE ALGORITMOS

Complexidade de tempo:

– Pode ser medida empiricamente:

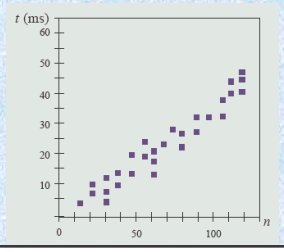
• Com funções para o cálculo do tempo. Exemplo:

```
#include <time.h>
time_t t1,t2;
time(&t1);

/*Operações*/

time(&t2);
double diferenca = difftime(t2,t1);
//diferença em segundos
```

TEMPO DE EXECUÇÃO DE ALGORÍTMOS

Estudo experimental

- escreva um programa que implemente o algoritmo;
- execute o programa com conjuntos de dados de vários tamanhos e composições;
- use um método para medir o tempo de execução com exatidão;
- os resultados devem ser parecidos com este



TEMPO DE EXECUÇÃO DE ALGORÍTMOS

- Estudos experimentais possuem várias limitações:
  - é preciso implementar e testar o algoritmo para determinar seu tempo de execução;
  - os experimentos só podem ser feito para um conjunto limitado de dados de entrada, assim os tempos de computação resultantes podem não ser indicativos dos tempos de execução para entradas não incluídas no experimento;
  - para comparar dois algoritmos, devem ser utilizados o mesmo ambiente de hardware e software.

TEMPO DE EXECUÇÃO DE ALGORÍTMOS

- iremos agora apresenta um metodologia geral para analisar o tempo de computação de algoritmos que
  - utiliza a descrição de algoritmos em alto-nível ao invés de testar uma de suas implementações
  - leva em consideração todas as possíveis entradas;
  - permite avaliar a eficiência de qualquer algoritmo de uma maneira que é independente dos ambientes de hardware e software.

COMPLEXIDADE DE ALGORITMOS

Uma boa idéia é estimar a eficiência de um algoritmo em função do tamanho do problema

- Em geral, assume-se que "*n*" é o tamanho do problema, ou número de elementos que serão processados
- E calcula-se o número de operações que serão realizadas sobre os *n* elementos

ANÁLISE ASSINTÓTICA

- Deve-se preocupar com a eficiência de algoritmos quando o tamanho de *n* for grande
- Definição: *a eficiência assintótica de um algoritmo descreve a eficiência relativa dele quando n torna-se grande*
- Portanto, para comparar 2 algoritmos, determinam-se as taxas de crescimento de cada um: o algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande

ANÁLISE ASSINTÓTICA

Procedimento

1) Escreve o algoritmo em pseudo-código;

2)Conta operações primitivas (computações de baixo nível que podem ser consideras em tempo constante de execução);

3) Análise a complexidade do algoritmo usando a notação Big-Oh.

Algumas notações

Notações que usaremos na análise de algoritmos

• $f(n) = O(g(n))$ (lê-se *big-oh*, *big-o* ou "da ordem de") se existirem constantes c e n_0 tal que $f(n) \leq c*g(n)$ quando $n \geq n_0$

- A taxa de crescimento de $f(n)$ é menor ou igual à taxa de $g(n)$

• $f(n) = \Omega(g(n))$ (lê-se "ômega") se existirem constantes c e n_0 tal que $f(n) \geq c*g(n)$ quando $n \geq n_0$

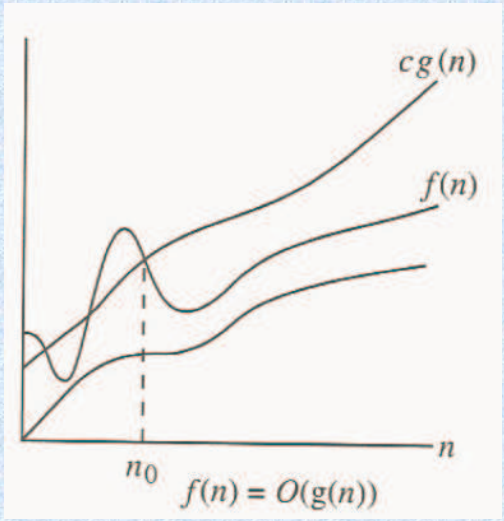
- A taxa de crescimento de $f(n)$ é maior ou igual à taxa de $g(n)$

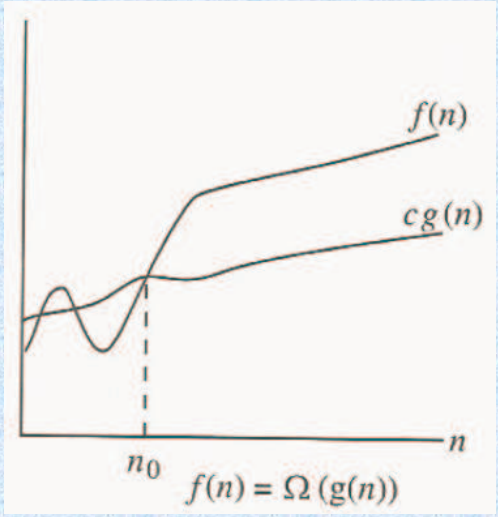
Algumas notações

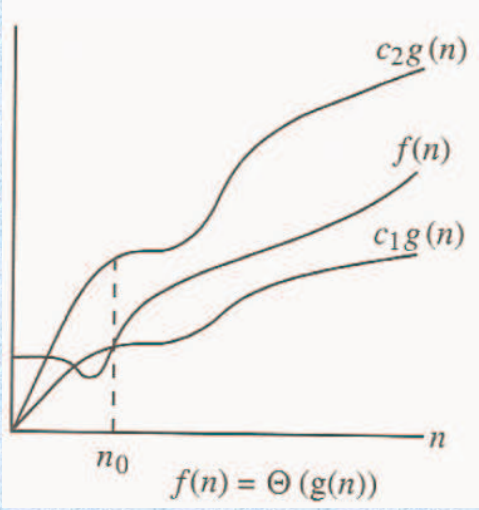
Notações que usaremos na análise de algoritmos

• $f(n) = \Theta(g(n))$ (lê-se "theta") se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

- A taxa de crescimento de $f(n)$ é igual à taxa de $g(n)$


cg(n)
f(n)
n
n0
f(n) = O(g(n))


f(n)
cg(n)
n
n0
f(n) = Ω (g(n))



Algumas considerações

• Ao dizer que T(n) = $O(f(n))$, garante-se que T(n) cresce numa taxa não maior do que $f(n)$, ou seja, $f(n)$ é seu limite superior

• Ao dizer que T(n) = $\Omega(f(n))$, tem-se que $f(n)$ é o limite inferior de T(n).

Exemplos

• Se $f(n)=n^2$ e $g(n)=2n^2$, então essas duas funções têm taxas de crescimento iguais. Portanto, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Taxas de crescimento

Algumas regras

- Se $T_1(n) = O(f(n))$ e $T_2(n) = O(g(n))$, então

  $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$

  $T_1(n) * T_2(n) = O(f(n) * g(n))$

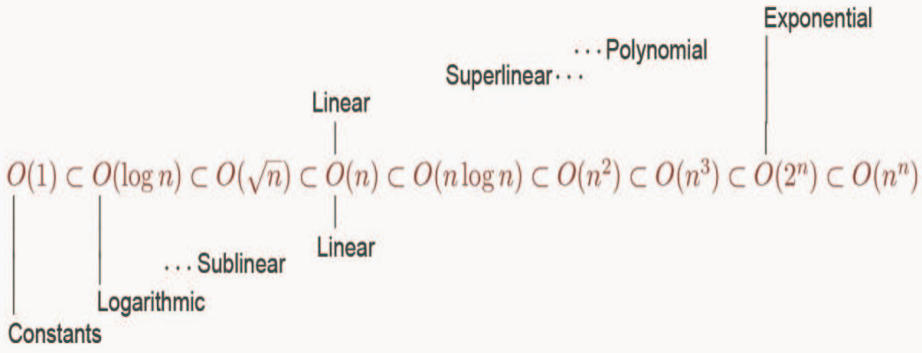
- Se T(x) é um polinômio de grau n, então

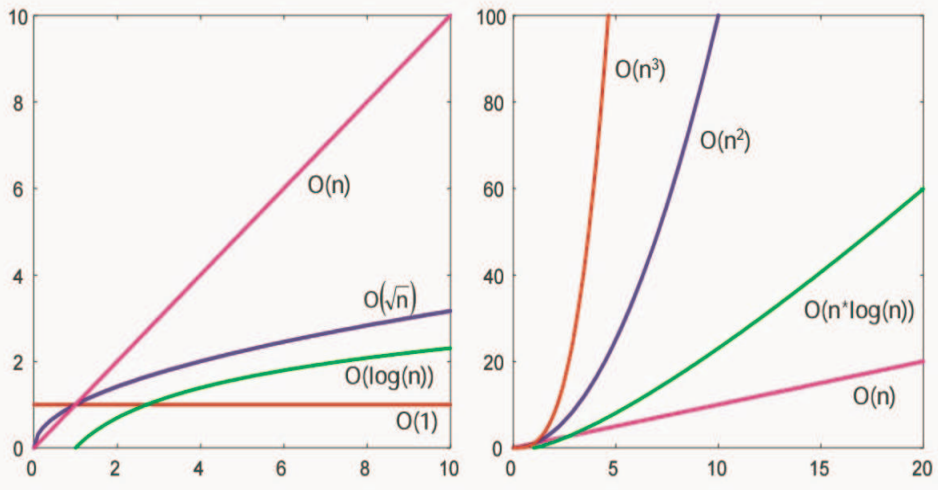
  $T(x) = O(x^n)$

Funções e Taxas de Crescimento

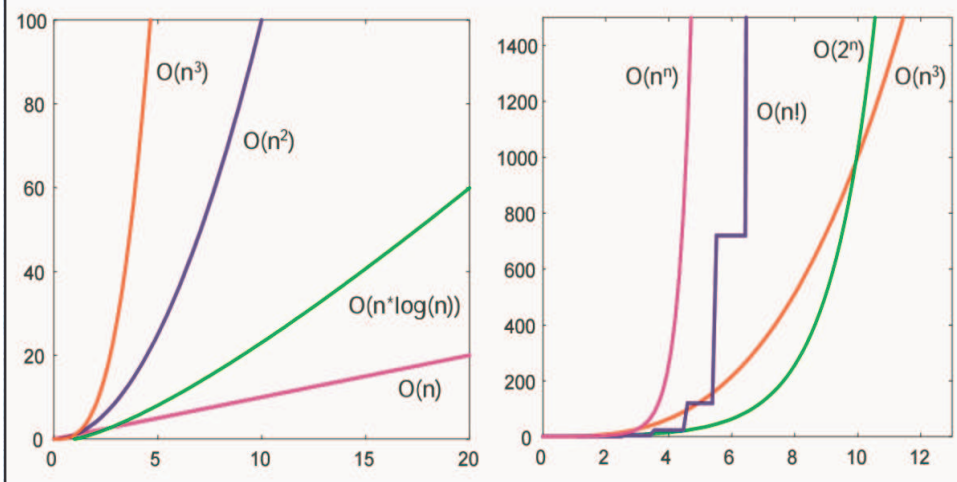
- Tempo constante: O(1) (raro)
- Tempo sublinear (log(n)): muito rápido (ótimo)
- Tempo linear: (O(n)): muito rápido (ótimo)
- Tempo $n\log n$: Comum em algoritmos de divisão e conquista.
- Tempo polinomial n^k : Freqüentemente de baixa ordem ($k \leq 10$), considerado eficiente.
- Tempo exponencial: 2^n , $n!$, n^n considerados intratáveis

Funções e Taxas de Crescimento





sublinear, linear and superlinear



superlinear, polynomial and exponential

Maximum size of a problem, n, that can be solved by several algorithms and computers with different processing performance:

Temporal Cost	1 step = 1 ms			1 step = 0.1 ms (10 times faster)			
	1 sec	1 min	1 hour	1 sec	1 min	1 hour	$n'=f(n)$
$\log_2 n$	$\approx 10^{330}$	$\approx 10^{2\cdot 10^4}$	$\approx 10^{10^{16}}$	$\approx 10^{3\cdot 10^3}$	$\approx 10^{2\cdot 10^5}$	$\approx 10^{10^{17}}$	n^{10}
n	1 000	$6\cdot 10^4$	$3{,}6\cdot 10^6$	10^4	$6\cdot 10^5$	$3{,}6\cdot 10^7$	$10\,n$
$n\log_2 n$	141	4 896	$2\cdot 10^5$	1 003	$4\cdot 10^4$	$1{,}7\cdot 10^6$	$\approx 9\,n$
n^2	32	245	1 897	100	775	6 000	$3{,}16\,n$
2^n	10	16	22	13	19	25	$n+3$

Execution times of a machine that executes 10^9 steps by second (~ 1 GHz), as a function of the algorithm cost and the size of input n:

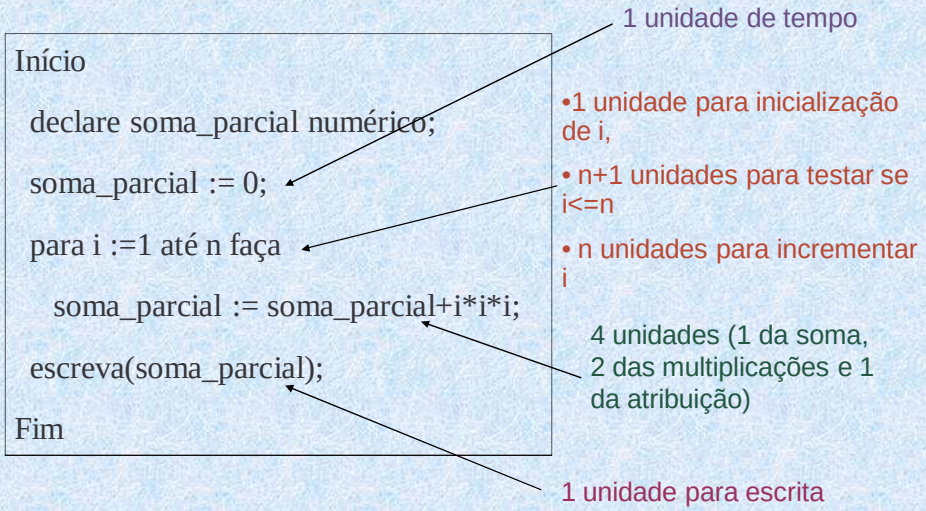
Size	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10	3.322 ns	10 ns	33 ns	100 ns	1 μs	1 μs
20	4.322 ns	20 ns	86 ns	400 ns	8 μs	1 ms
30	4.907 ns	30 ns	147 ns	900 ns	27 μs	1 s
40	5.322 ns	40 ns	213 ns	2 μs	64 μs	18.3 min
50	5.644 ns	50 ns	282 ns	3 μs	125 μs	13 days
100	6.644 ns	100 ns	664 ns	10 μs	1 ms	$40 \cdot 10^{12}$ years
1000	10 ns	1 μs	10 μs	1 ms	1 s	
10000	13 ns	10 μs	133 μs	100 ms	16.7 min	
100000	17 ns	100 μs	2 ms	10 s	11.6 days	
1000000	20 ns	1 ms	20 ms	16.7 min	31.7 years	

Calculando o tempo de execução

• Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

```
Início
  declare soma_parcial numérico;
  soma_parcial := 0;
  para i :=1 até n faça
     soma_parcial := soma_parcial+i*i*i;
  escreva(soma_parcial);
Fim
```

Calculando o Tempo de Execução



Custo total: 6n + 4 = O(n)

Calculando o Tempo de Execução

- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos
- No exemplo anterior
  - A linha “soma_parcial := 0” é insignificante em termos de tempo
  - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha “soma_parcial := soma_parcial+i*i*i”
  - O que realmente dá a grandeza de tempo desejada é a repetição na linha “para i := 1 até n faça”

Em geral, não consideramos os termos de ordem inferior da complexidade de um algoritmo, apenas o termo predominante.

Exemplo: Um algoritmo tem complexidade $T(n) = 3n^2 + 100n$. Nesta função, o segundo termo tem um peso relativamente grande, mas a partir de $n_0 = 11$, é o termo n^2 que "dá o tom" do crescimento da função: uma parábola. A constante 3 também tem uma influência irrelevante sobre a taxa de crescimento da função após um certo tempo. Por isso dizemos que este algoritmo é da ordem de n^2 ou que tem complexidade $O(n^2)$.

Regras para o cálculo

- Repetições

O tempo de execução de uma repetição é o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

Regras para o cálculo

- Repetições aninhadas
  - A análise é feita de dentro para fora
  - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
  - O exemplo abaixo é $O(n^2)$

```
para i := 0 até n faça
   para j := 0 até n faça
      faça k := k+1;
```

Regras para o cálculo

- Comandos consecutivos
  - É a soma dos tempos de cada um bloco, o que pode significar o máximo entre eles
  - O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser O(n)

```
para i := 0 até n faça
   k := 0;
para i := 0 até n faça
   para j := 0 até n faça
      faça k := k+1;
```

Regras para o cálculo

- Se... então... senão

  - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao então e os comandos relativos ao senão

  - O exemplo abaixo é O(n)

```
se i < j
então i := i+1
senão para k := 1 até n faça
                i := i*k;
```

Exercício

- Estime quantas unidades de tempo são ecessárias para rodar o algoritmo abaixo

```
Início
   declare i e j numéricos;
   declare A vetor numérico de n posições;
   i := 1;
   enquanto i <= n faça
      A[i] := 0;
      i := i+1;
   para i := 1 até n faça
      para j := 1 até n faça
         A[i] := A[i]+i+j;
Fim
```

Regras para o cálculo

- Chamadas a sub-rotinas

  Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

Regras para o cálculo

- Sub-rotinas recursivas
  - Análise de recorrência
  - *Recorrência: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores*
  - Caso típico: algoritmos de **dividir-e-conquista**r, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original

Regras para o cálculo

- Exemplo de uso de recorrência
  - Calculo Fatorial n!

```
sub-rotina fat(n: numérico)
início
   declare aux numérico;
   aux := 1
   se n = 1
   então aux := 1
   senão aux := n*fat(n-1);
fim
```

Regras para o cálculo

```
sub-rotina fat(n: numérico)
início
   declare aux numérico;
   aux := 1
   se n = 1
   então aux := 1
   senão aux := n*fat(n-1);
fim
```

$$T(n) = c+T(n-1)$$
$$= 2c+T(n-2)$$
$$= \ldots$$
$$= nc +T(1)$$
$$= O(n)$$

UMA RÁPIDA REVISÃO DE MATEMÁTICA

Logaritmos e Expoentes

- Propriedades de logaritmos

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b x/y = \log_b x - \log_b y$$

$$\log_b x^\alpha = \alpha \log_b x$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$b^{\log_c a} = a^{\log_c b}$$

- Propriedades de expoente

$$a^{b+c} = a^b a^c$$

$$\left(a^b\right)^c = a^{bc}$$

$$\frac{a^b}{a^c} = a^{b-c}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c \log_a b}$$

UMA RÁPIDA REVISÃO DE MATEMÁTICA

- Chão (floor)

  $\lfloor x \rfloor$= ao maior inteiro menor que x

- Teto (ceiling)

  $\lceil x \rceil$ = ao menor inteiro menor que x

- Somatória

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \ldots + f(t-1) + f(t)$$

UMA RÁPIDA REVISÃO DE MATEMÁTICA

Expressões geométricas $f(i) = a^i$

Dado um número inteiro $n > 0$ e um número real $0 < a \neq 1$,

$$\sum_{i=0}^{n} a^i = a^0 + a^1 + a^2 + ... + a^n = \frac{1-a^{n+1}}{1-a}$$

A progressão geométrica mostra um crescimento exponencial

UMA RÁPIDA REVISÃO DE MATEMÁTICA

Progressões aritméticas

Um exemplo,

$$\sum_{i=1}^{n} i = 1+2+3+...+n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{1}{3}n(n+1)\left(n+\frac{1}{2}\right)$$