

Introdução a Análise Assintótica

Vamos falar um pouco das notações que usamos na análise assintótica.

Big O, Theta θ , Ômega Ω

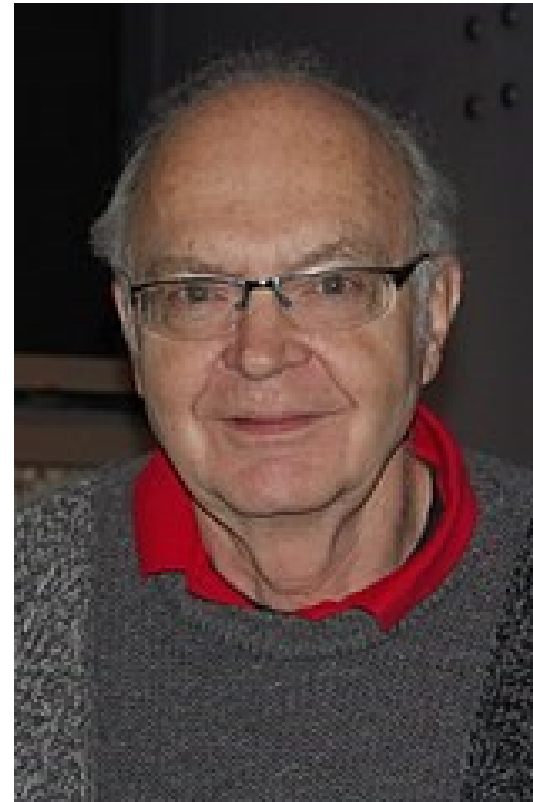
Notação Big O.

A notação O foi originalmente postulada em 1894 por um matemático alemão chamado Paul Bachmann.



Bachmann estudou matemática na Universidade de Berlim e obteve seu doutorado em 1862, com tese sobre a teoria de grupos.

Em 1976 Donald Knuth escreve uma carta à SIGACT propondo que os editores de periódicos e artigos adotem a notação O como forma de análise de algoritmos em suas publicações.



Donald Ervin Knuth nasceu em 10 de janeiro de 1938, é um cientista da computação americano.

(matemático e professor emérito da Universidade de Stanford)

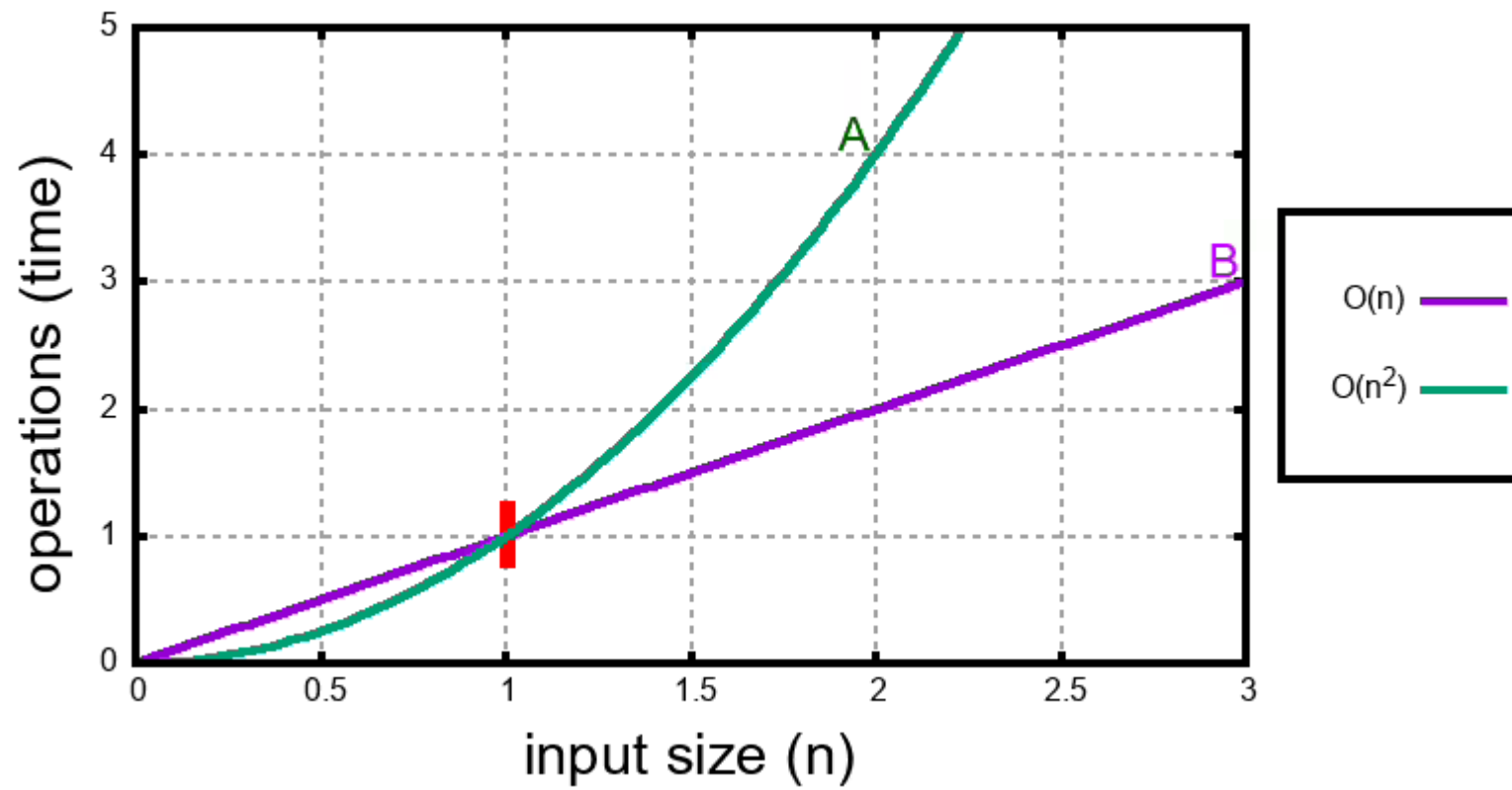
“Knuth foi chamado de “pai da análise de algoritmo”

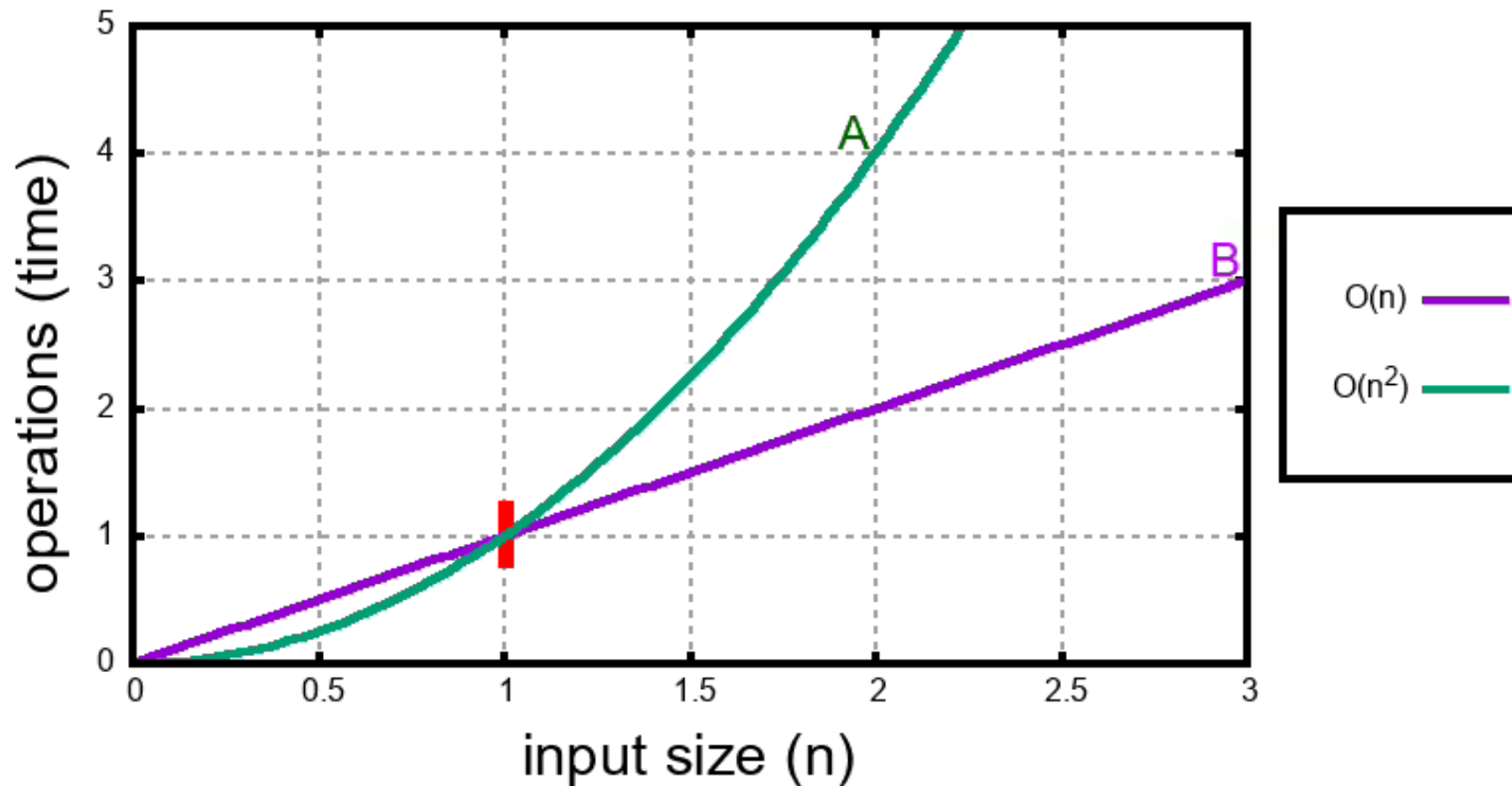
Vamos começar nosso estudo através de um exemplo hipotético.

Considere que você quer avaliar dois algoritmos para envio de arquivos na Internet.

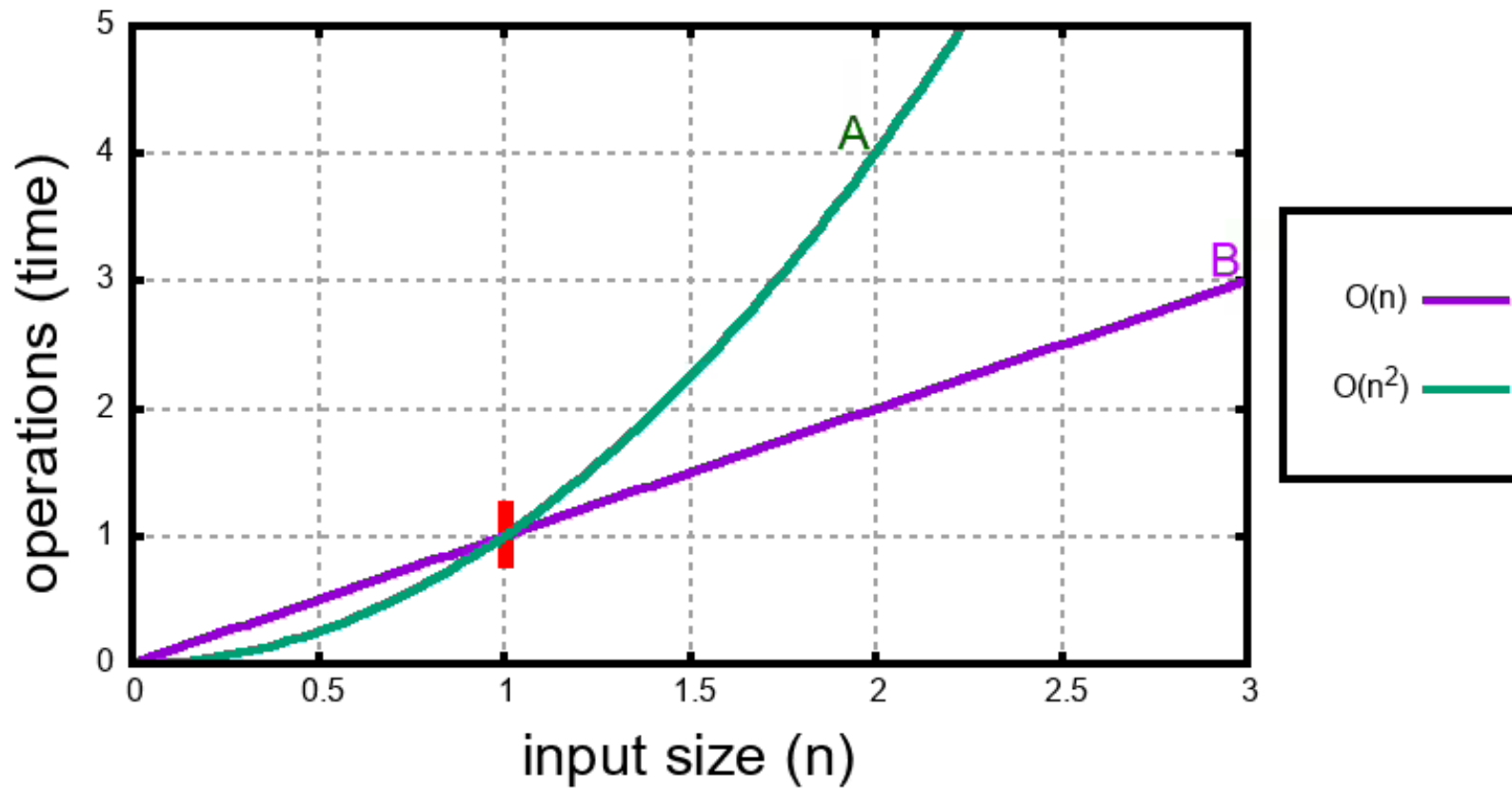
Vamos chamá-los de algoritmo A e B.

Vamos assumir também em nosso exemplo que A tem complexidade $O(n^2)$ e B complexidade $O(n)$.



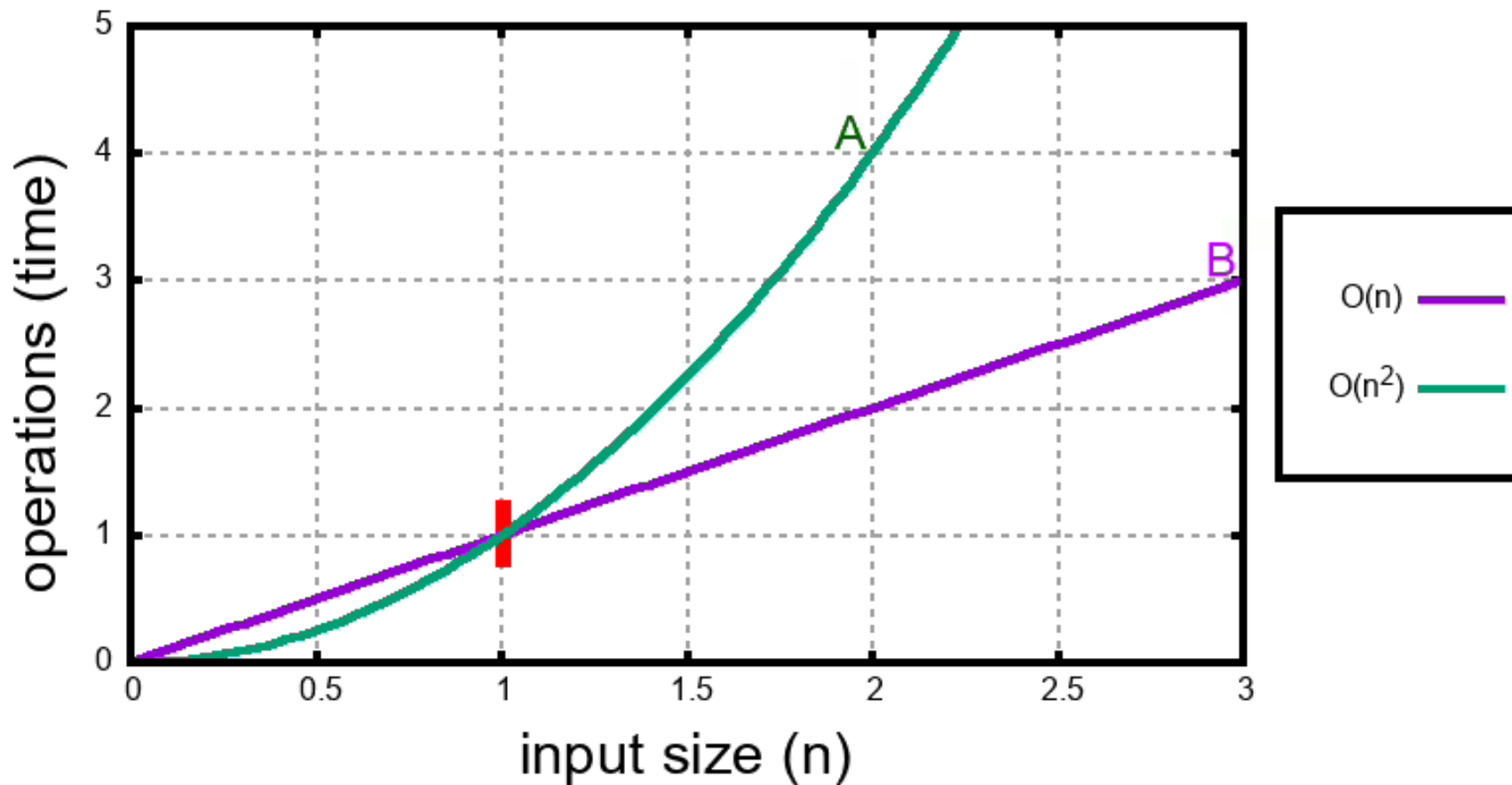


Na figura podemos notar algumas coisas. Considere que no eixo x temos o tamanho do arquivo em *gigabytes*. Esse arquivo é o parâmetro de entrada de nossos algoritmos. Já no eixo y temos o número de operações necessárias para fazer o envio deste arquivo



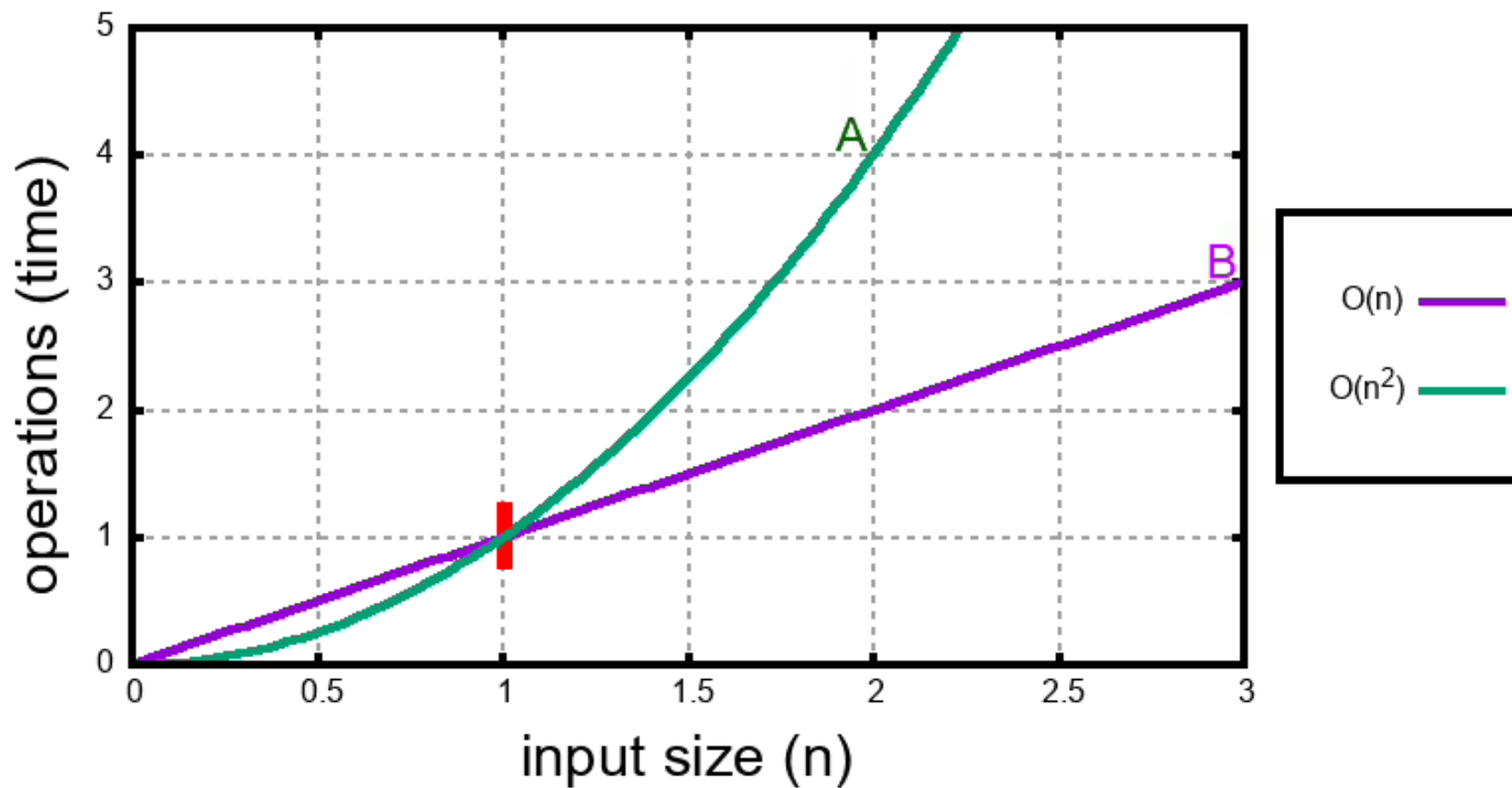
Logo, podemos notar então que:

à medida que tamanho do arquivo aumenta, o algoritmo A explode de maneira polinomial em número de operações.

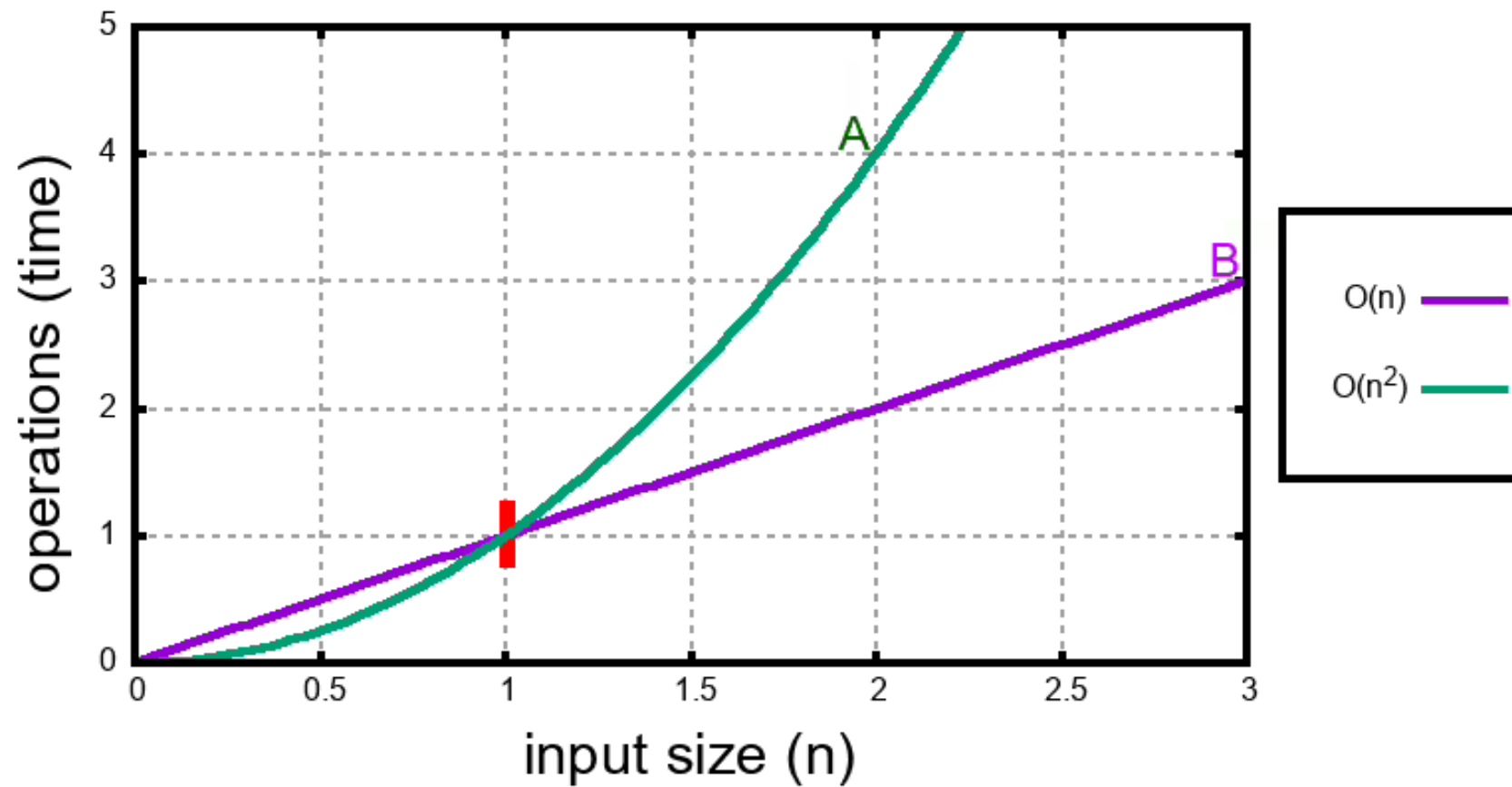


Este algoritmo segue uma classe, uma ordem de funções que se comportam/crescem como uma quadrática.

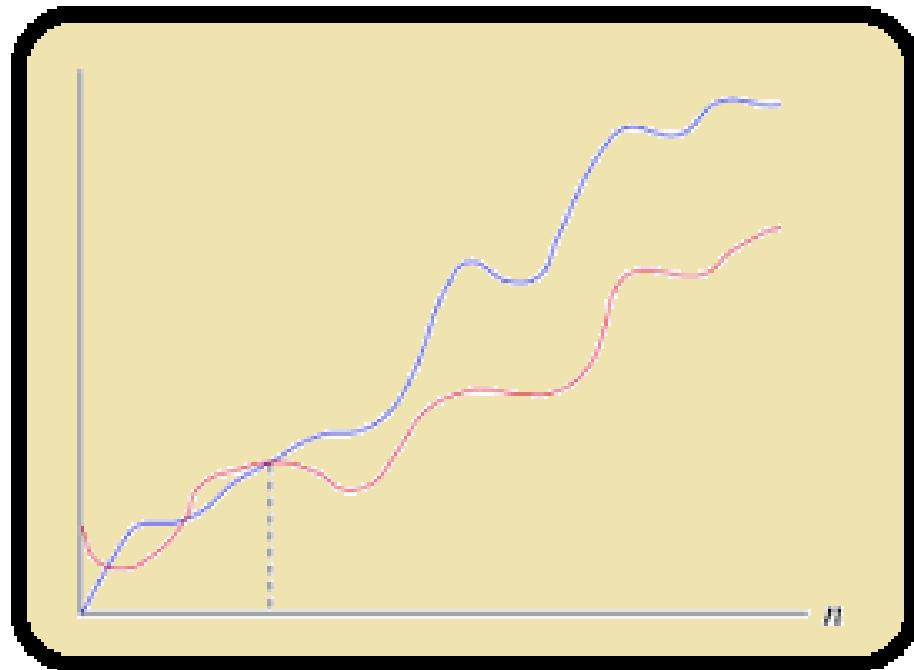
Já o algoritmo B demonstra um crescimento linear, pois à medida que o tamanho do arquivo de entrada cresce, o número de operações cresce na mesma proporção. Ou seja, linearmente.



Outra coisa essencial a se notar neste exemplo é a marcação em vermelho no ponto (1,1). Observe que, se alguém utilizar e comparar estes programas utilizando sempre arquivos menores do que 1 Gb, o algoritmo A será considerado mais eficiente, pois executa menos operações, segundo este gráfico.



Esse detalhe é importante pois isto é fundamental para a análise do comportamento assintótico das funções que representam a execução destes algoritmos.



Notação O

Definindo a notação O

Seja:

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

Se existirem uma constante c e um valor n_0 tal que:

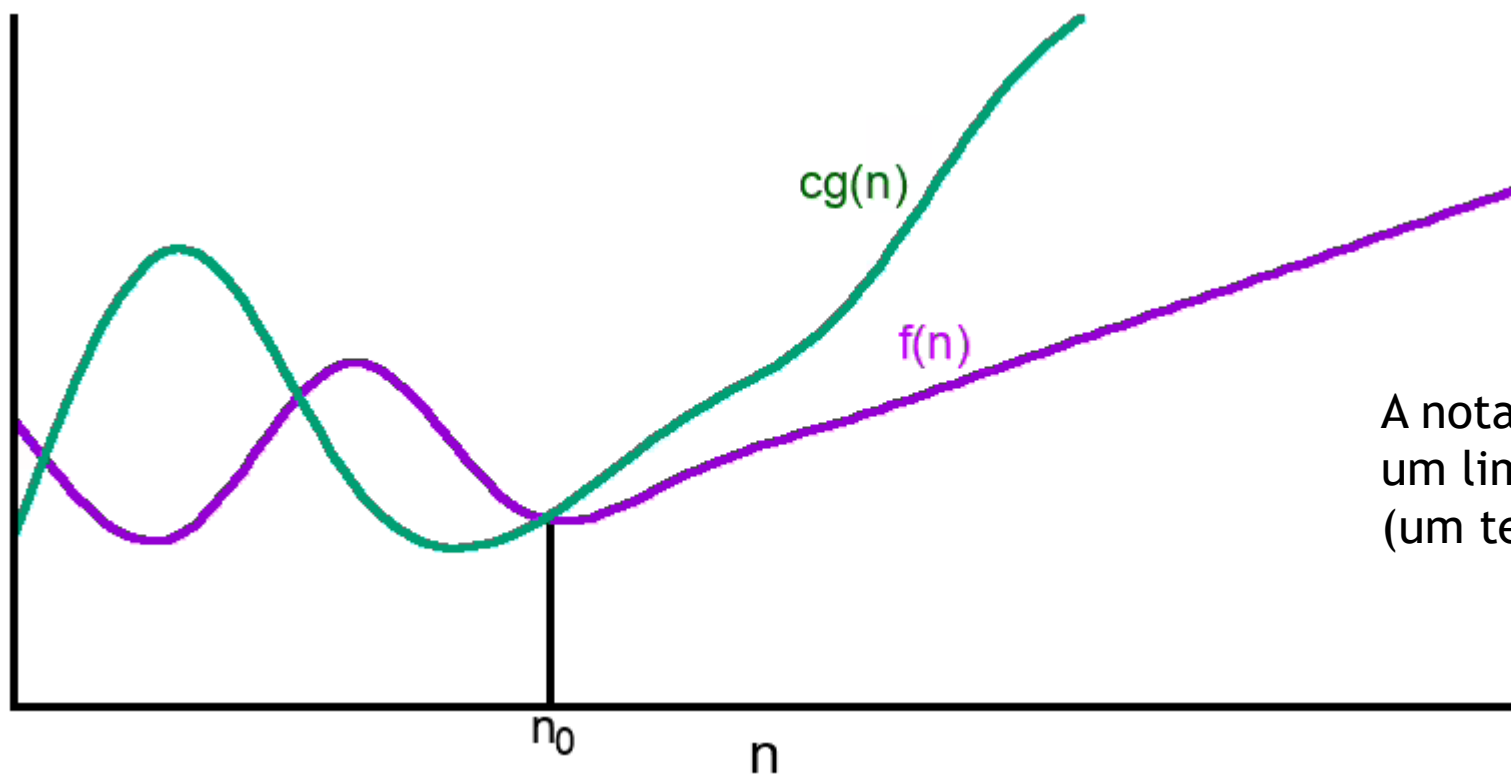
$$0 \leq f(n) \leq cg(n), \forall n \geq n_0$$

Podemos ler da seguinte forma: A função $f(n)$ cresce no máximo na ordem de $g(n)$ para qualquer valor da entrada n que seja maior ou igual a n_0 .

Ou seja, $g(n)$ é um limite superior para o crescimento de $f(n)$.

$$0 \leq f(n) \leq cg(n), \forall n \geq n_0$$

Utiliza-se uma constante c positiva de modo a poder deslocar a função $g(n)$ e mesmo assim ela ser um limite superior de $f(n)$ para todos os valores maiores do que n_0 .



A notação O define
um limite superior
(um teto).

Em resumo: $f(n)$ não cresce mais rápido do que $g(n)$.

Exemplo

Mostre $5n + 3 = O(n)$

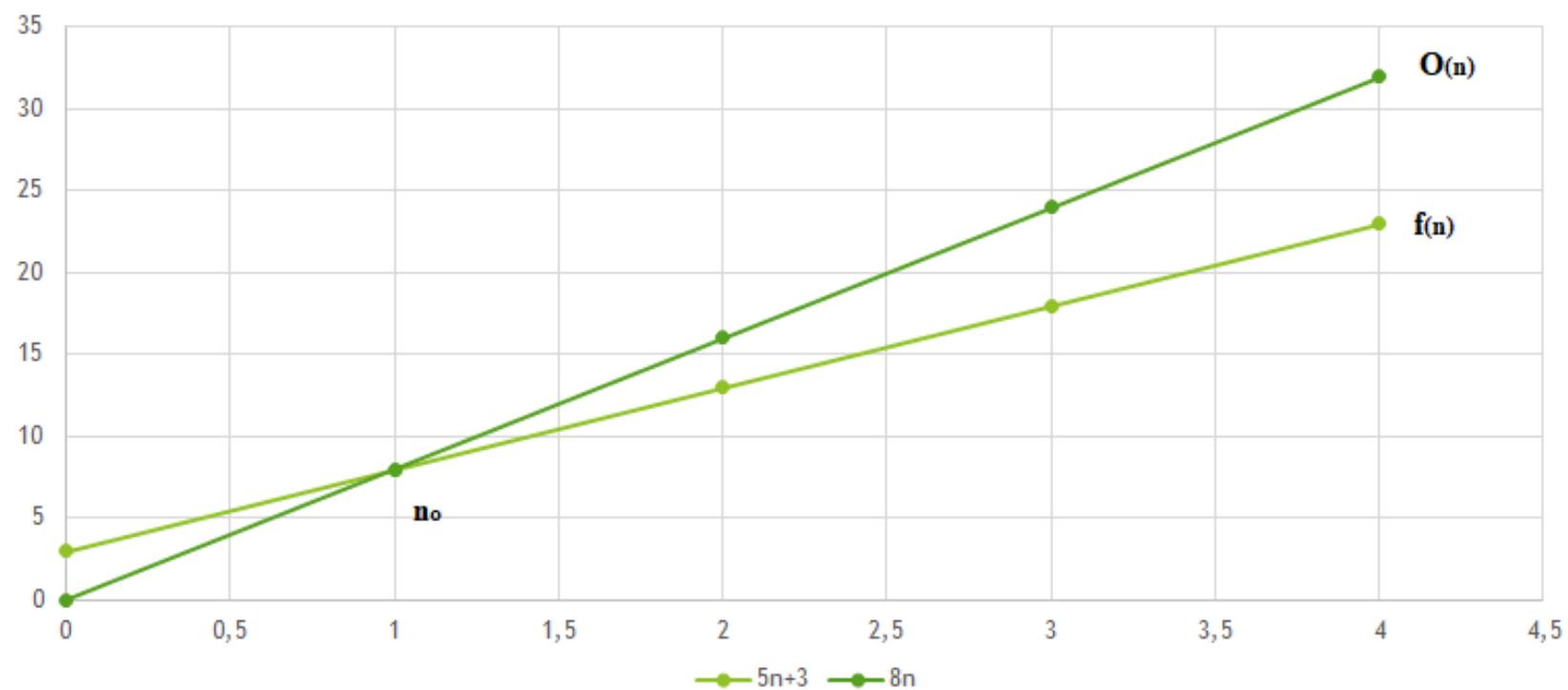
$$0 \leq f(n) \leq cg(n), \forall n \geq n_0$$

Solução:

$$5n + 3 \leq c.n$$

$$3 \leq cn - 5n$$

$$3 \leq (c - 5)n$$



Observe que para $c = 8$ teremos $3 \leq 8n$, então teremos $\forall n \geq n_0$ onde $n_0 = 1$

RELEMBRANDO:

A MEDIDA DE TEMPO: OPERAÇÕES

O que exatamente é o eixo y quando analisamos o comportamento das funções que representam os algoritmos?

Estamos falando de operações. Operações como uma soma, um *if* dentro de um programa.

É muito comum encontrarmos a seguinte notação de $O(1)$.

Representa um tempo constante.

No entanto, quando estamos falando de instruções de máquina olhamos o número de ciclos de CPU para execução de uma determinada operação.

Então um *if*, uma soma, ou carregar um dado da memória principal, cada uma dessas operações pode ter custo diferente em cada processador em que é executado.

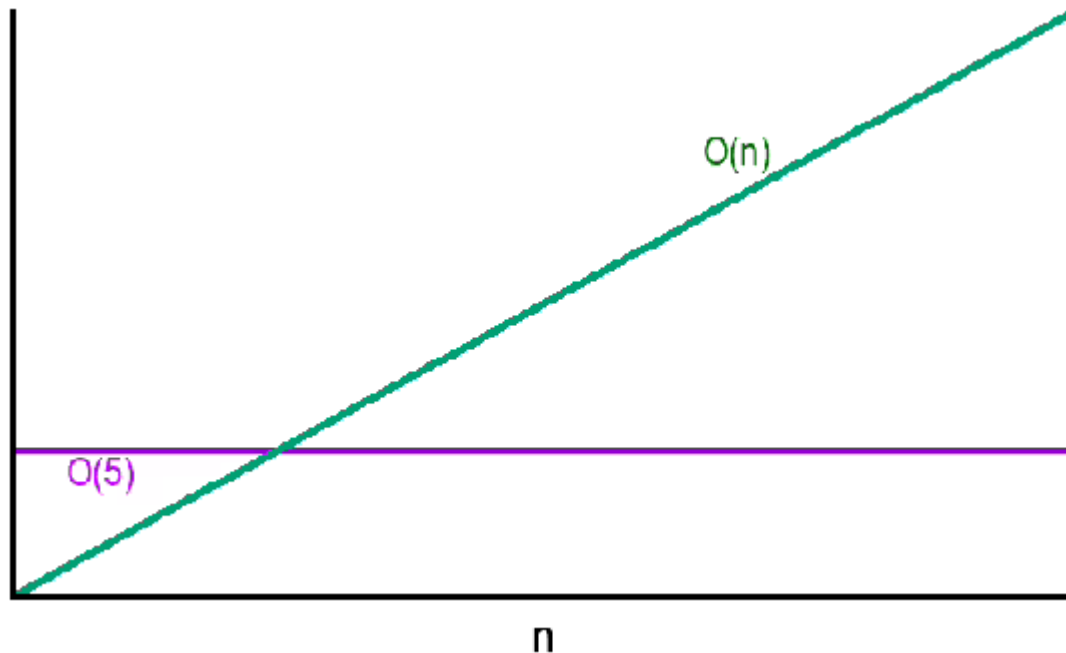
Assim, por mais que às vezes, uma dessas operações pode custar 5 ciclos e outra 1 ciclo, todas elas são consideradas constantes. Logo, ditas $O(1)$.

Mas não tem problema dizer $O(5)$.

No entanto, se eu escrevo um *for loop* de n repetições, sendo $n=10$, independente do corpo do *loop* o programa irá executar n vezes. Logo, $O(n)$.

O que é importante notar?

Independente da entrada n do programa, uma operação constante será sempre constante mas um *loop* por exemplo sempre irá crescer em função do tamanho dessa entrada. Ou seja, linearmente à n . A imagem abaixo mostra essas duas complexidades:



Incrementalmente, vamos olhar um programa com dois *loops* aninhados e interpretar a complexidade inerente a este algoritmo e como esta interpretação seguirá a interpretação do exemplo anterior e como ela demonstra a intuição por trás da análise de complexidade.


```
Int some_function (int n) {  
...  
    for (int i=0; i < n; i ++ ) {  
        for ( int j = 0; j < n; j++ ) {  
            // do some operation  
        }  
    }  
    ...  
}
```

Este algoritmo está diretamente limitado à entrada n .
Temos dois *loops*. Para cada iteração i do *loop* externo
são executadas outras n iterações j .

Ou seja, Se n for 8, temos que esse programa irá
rodar n vezes n iterações. 8 vezes o *loop* externo, 8
vezes do *loop* interno, totalizando 64 iterações.

No entanto, n é uma variável de entrada. E se n for 1 milhão?
Ou se ele for 2, ou 6 bilhões?
Como estimar o tempo de execução total deste programa?

É aí que a Notação O nos ajuda a definir o crescimento de uma função que representa um algoritmo independente de qual o tamanho da entrada do programa. Tomando como exemplo o algoritmo apresentado acima temos:

$$n \times n = n^2$$

Como a notação O é o limite superior, podemos afirmar que para este exemplo que, no pior caso, este algoritmo irá crescer na ordem de n^2 .

Vamos chamar o nosso programa de $f(x)$.

$$\therefore f(x) = O(n^2).$$

Assim, seguindo esta intuição você poderá analisar a complexidade de diversos algoritmos para inferir como o tempo de execução dos programas se comportará em função do tamanho de suas entradas.

Uma função ou algoritmo cheio de condicionais e regras que controlam sua execução pode retornar (chamar o *return* da função) antes de executar por exemplo todas as iterações um *loop*.

Ou seja, nem sempre um programa irá recair na complexidade do seu pior caso (limite superior).

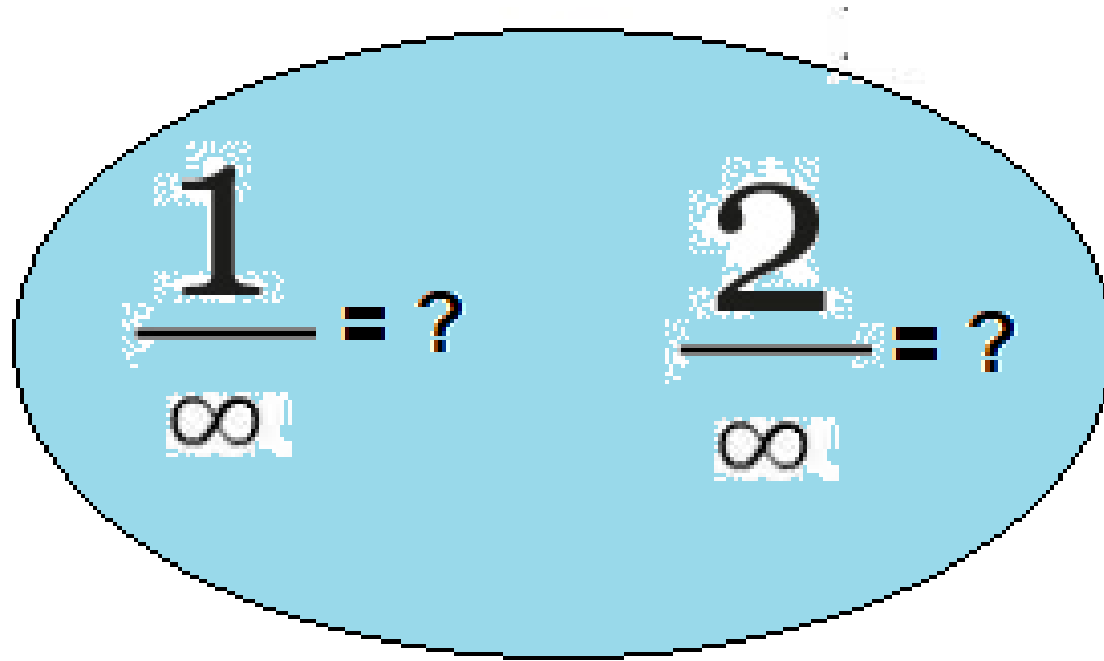
Mas é exatamente isto que a Notação O nos entrega, a estimativa do limite superior

O que pode inclusive ser o pior caso da execução de um algoritmo.

Logo, podemos dizer que a execução daquele algoritmo nunca será pior do que a complexidade extraída pela Notação O .

Se estamos falando de quantidade de operações, porquê dizemos complexidade de tempo de execução?

Bem, quando falamos que uma operação é na verdade computada em ciclos de CPU dependendo do processador subjacente, na prática, estamos falando de tempo. Ciclos de CPU são medidos em nanosegundos. Logo, uma medida de tempo.


$$\frac{1}{\infty} = ? \quad \frac{2}{\infty} = ?$$

Relembrando algumas operações matemáticas.

Exemplos

Considere a expressão: $\frac{5}{n}$ onde n vai sempre aumentar.

a) $\frac{5}{2} =$

b) $\frac{5}{10} =$

c) $\frac{5}{100} =$

d) $\frac{5}{10000} =$

e) E se n tende para infinito, (∞ simbolo de infinito), para qual valor a divisão tende?

$$\frac{5}{\infty} = ?$$

Exercício

Mostre $5n^2 + 3n + 4 = O(n^2)$

Solução

$$5n^2 + 3n + 4 \leq C \cdot n^2$$

Dividimos
Tudo por
 n^2

$$5 \frac{m^2}{n^2} + 3 \frac{m}{n^2} + \frac{4}{n^2} \leq C \cdot \frac{m^2}{n^2}$$

$$5 + \frac{3}{n} + \frac{4}{n^2} \leq C$$

$$5 + \frac{3}{\infty} + \frac{4}{\infty} \leq C$$

$$5 + 0 + 0 \leq C \Rightarrow$$

$$\boxed{5 \leq C} \Rightarrow C \geq 5$$

Atende.

Lembre que
n tende a
ser muito
grande
 $n \rightarrow \infty$

n_0

n	$5n^2+3n+4$	$6n^2$
0	4	0
0,5	6,75	1,5
1	12	6
2	30	24
3	58	54
4	96	96
5	144	150
6	202	216

Gráfico

