Análise de Complexidade

Parte 3

Prof. Kennedy Reurison Lopes

July 12, 2023

Complexidade Assintótica

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Notação	Descrição simplificada

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Notação	**Descrição simplificada**
$O(n)$	Limitador Estrito Superior

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Notação	Descrição simplificada
$O(n)$	Limitador Estrito Superior
$\Omega(n)$	Limitador Estrito Inferior

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Notação	**Descrição simplificada**
$O(n)$	Limitador Estrito Superior
$\Omega(n)$	Limitador Estrito Inferior
$\Theta(n)$	Limitador Central

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Notação	**Descrição simplificada**
$O(n)$	Limitador Estrito Superior
$\Omega(n)$	Limitador Estrito Inferior
$\Theta(n)$	Limitador Central
$o(n)$	Limitador Não Estrito Superior

Complexidade Assintótica

- Na análise de algoritmos, geralmente usa-se a complexidade assintótica, analisando o algoritmo para n tendendo a infinito;
- Nesse caso, despreza-se constantes e termos de menor crescimento;
- Usa-se notações especiais para a complexidade assintótica.

Tipos de Notação:

Notação	**Descrição simplificada**
$O(n)$	Limitador Estrito Superior
$\Omega(n)$	Limitador Estrito Inferior
$\Theta(n)$	Limitador Central
$o(n)$	Limitador Não Estrito Superior
$\omega(n)$	Limitador Não Estrito Inferior

Notação O

Uma função $f(n)$ é dita como sendo $O(g(n))$ se existe uma constante positivas c para o qual:

$$0 < f(n) \leq c \times g(n)$$

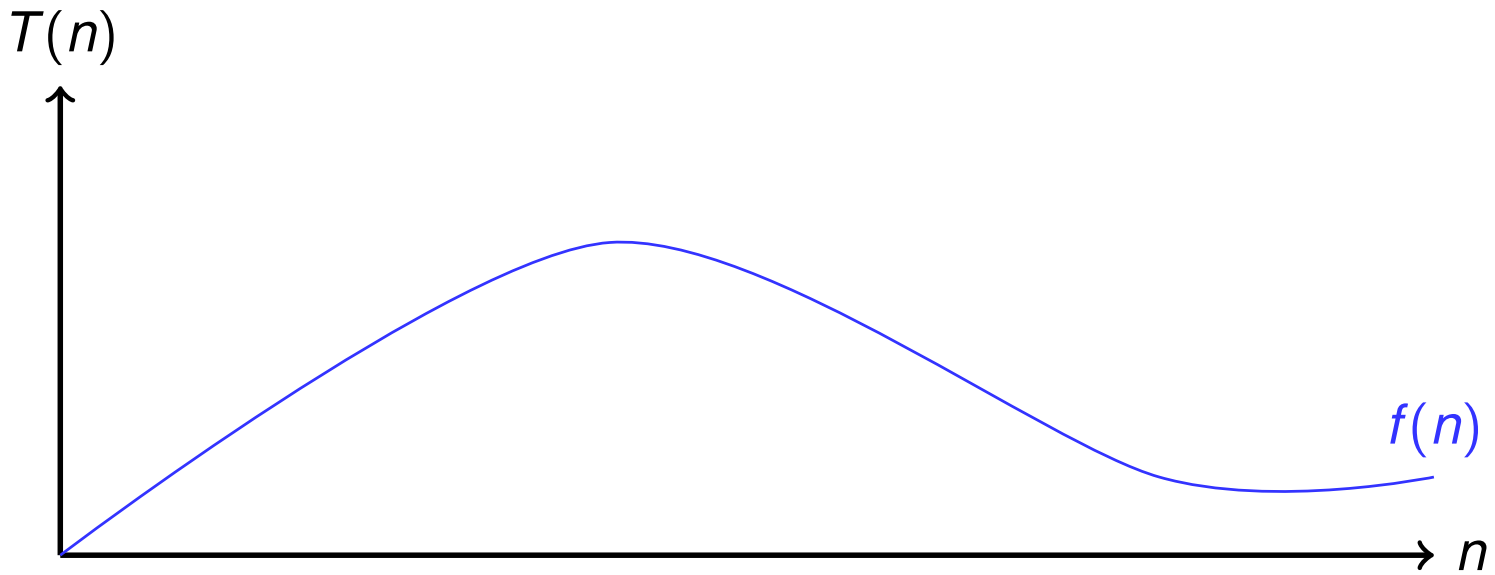
para todo $n > n_0$.

Notação O

Uma função $f(n)$ é dita como sendo $O(g(n))$ se existe uma constante positivas c para o qual:

$$0 < f(n) \leq c \times g(n)$$

para todo $n > n_0$.

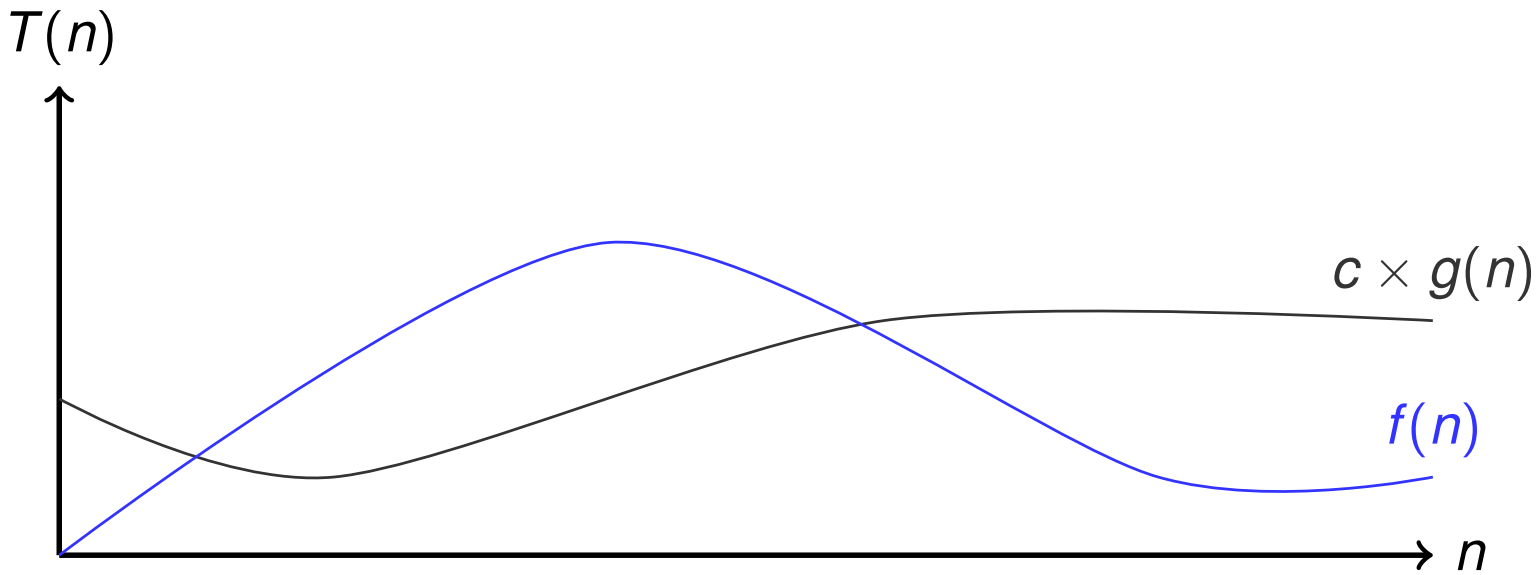


Notação O

Uma função $f(n)$ é dita como sendo $O(g(n))$ se existe uma constante positivas c para o qual:

$$0 < f(n) \leq c \times g(n)$$

para todo $n > n_0$.

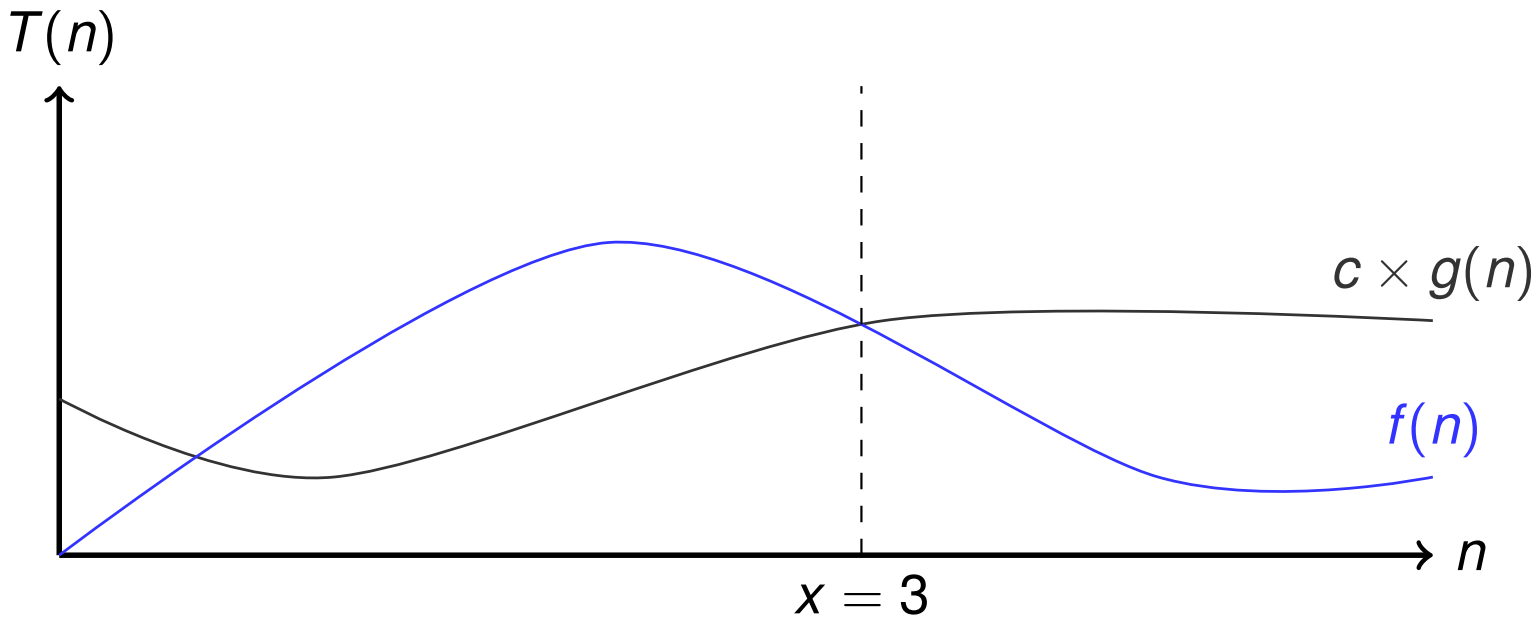


Notação O

Uma função $f(n)$ é dita como sendo $O(g(n))$ se existe uma constante positivas c para o qual:

$$0 < f(n) \leq c \times g(n)$$

para todo $n > n_0$.

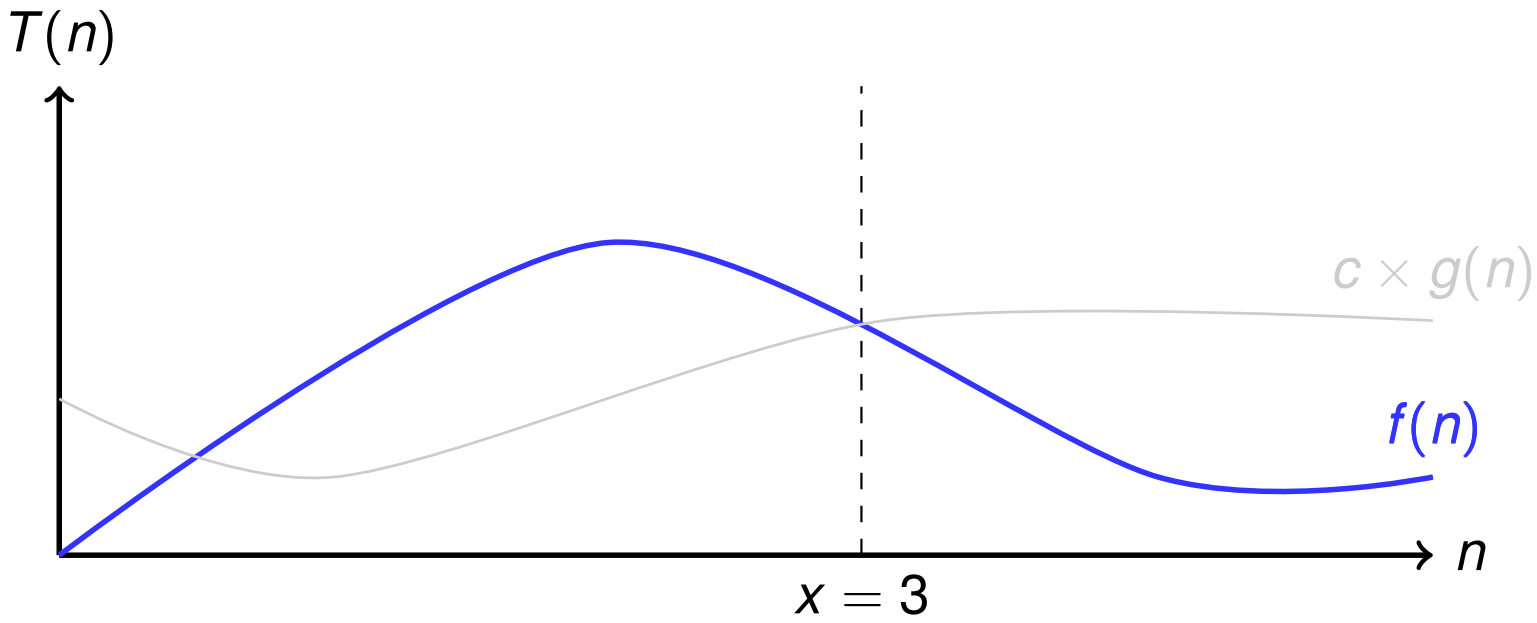


Notação O

Uma função $f(n)$ é dita como sendo $O(g(n))$ se existe uma constante positivas c para o qual:

$$0 < f(n) \leq c \times g(n)$$

para todo $n > n_0$.



Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

L2) 1

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

L2) 1

L3) $1 + n/2$

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

L2) 1

L3) $1 + n/2$

L4) $1 + (n/4)(n/2)$

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

L2) 1

L3) $1 + n/2$

L4) $1 + (n/4)(n/2)$

L5) $3(n/4)(n/2)$

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

L2) 1

L3) $1 + n/2$

L4) $1 + (n/4)(n/2)$

L5) $3(n/4)(n/2)$

L6) 1

Exemplo: Complexidade O

Verifique se o algoritmo é $O(n^2)$.

```
1 void funcao(int *v, int n){
2   int soma = 0;
3   for (int i=0; i<(n/2); i++)
4     for (int j=0; j<(n/4); j++)
5       soma += i + j;
6   return soma;
7 }
```

L2) 1

L3) $1 + n/2$

L4) $1 + (n/4)(n/2)$

L5) $3(n/4)(n/2)$

L6) 1

$$T(n) = 4 + \left(\frac{n}{2}\right) + \left(\frac{n}{2}\frac{n}{4}\right) + 3\left(\frac{n}{2}\frac{n}{4}\right)$$

$$T(n) = 4 + \frac{n}{2} + \frac{n^2}{2}$$

Exemplo: Complexidade O

$$T(n) = 4 + \frac{n}{2} + \frac{n^2}{2} = \frac{8 + n + n^2}{2}$$

$$0 < f(n) \leq c \times g(n)$$

$$0 < \frac{8 + n + n^2}{2} \leq c_1 \times n^2$$

A primeira inequação sempre será válida, basta então avaliar a segunda inequação:

Exemplo: Complexidade O

$$T(n) = 4 + \frac{n}{2} + \frac{n^2}{2} = \frac{8 + n + n^2}{2}$$

$$0 < f(n) \leq c \times g(n)$$

$$0 < \frac{8 + n + n^2}{2} \leq c_1 \times n^2$$

A primeira inequação sempre será válida, basta então avaliar a segunda inequação:

Exemplo: Complexidade O

$$T(n) = 4 + \frac{n}{2} + \frac{n^2}{2} = \frac{8 + n + n^2}{2}$$

$$0 < f(n) \leq c \times g(n)$$

$$0 < \frac{8 + n + n^2}{2} \leq c_1 \times n^2$$

A primeira inequação sempre será válida, basta então avaliar a segunda inequação:

$$\frac{n^2 + n + 8}{2} \leq c_1 n^2$$

$$n^2 + n + 8 \leq 2c_1 n^2$$

$$n^2(1 - 2c_1) + n + 8 \leq 0$$

Exemplo: Complexidade O

$$T(n) = 4 + \frac{n}{2} + \frac{n^2}{2} = \frac{8 + n + n^2}{2}$$

$$0 < f(n) \leq c \times g(n)$$

$$0 < \frac{8 + n + n^2}{2} \leq c_1 \times n^2$$

A primeira inequação sempre será válida, basta então avaliar a segunda inequação:

$$\frac{n^2 + n + 8}{2} \leq c_1 n^2$$

$$n^2 + n + 8 \leq 2c_1 n^2$$

$$n^2(1 - 2c_1) + n + 8 \leq 0$$

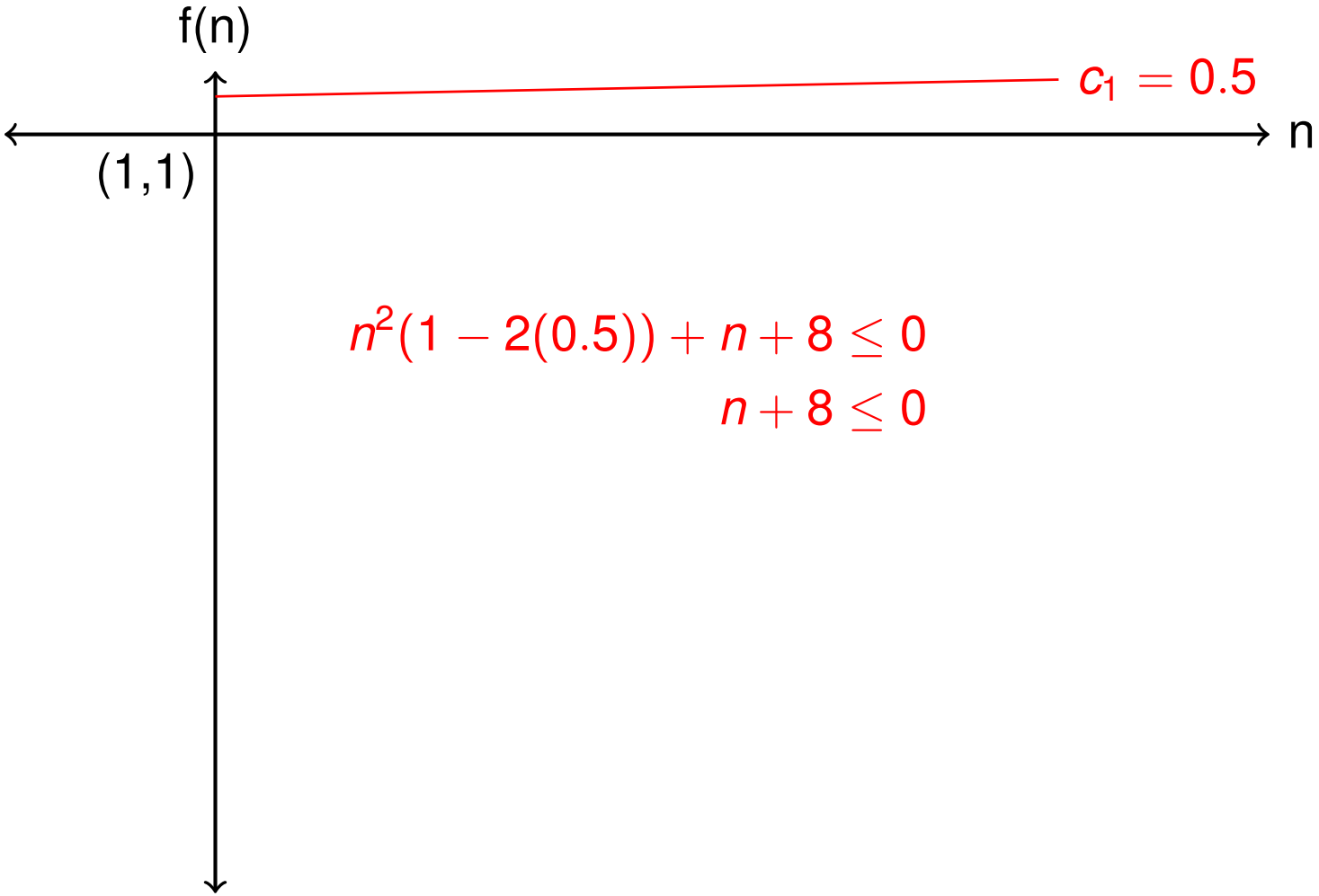
Os valores de c_1 indicarão em quais condições n permitirá que a inequação será verdadeira.

Análise de $n^2(1 - 2c_1) + n + 8 \leq 0$

Escolha de c_1:

Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:

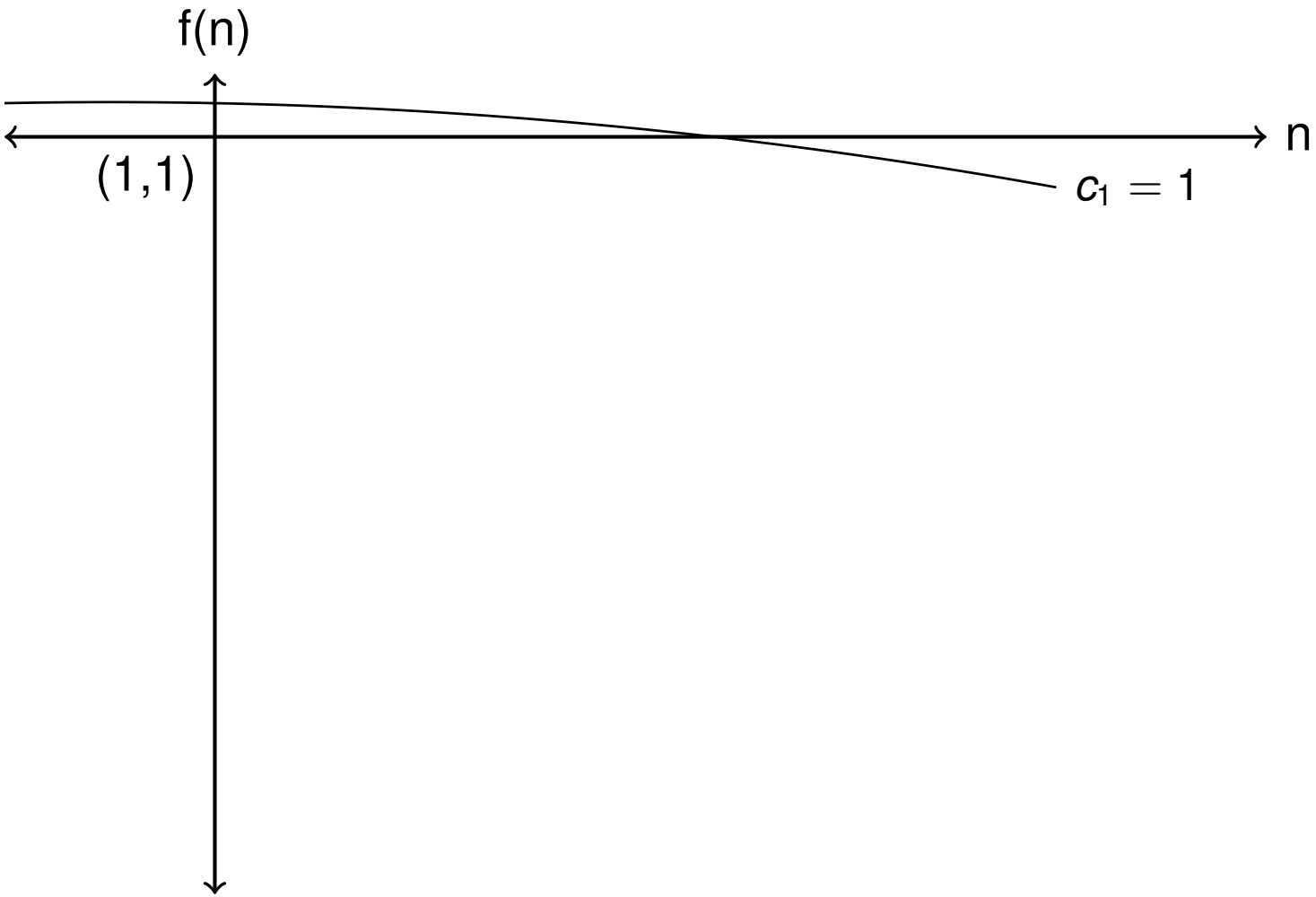


$$n^2(1-2(0.5))+n+8 \leq 0$$

$$n+8 \leq 0$$

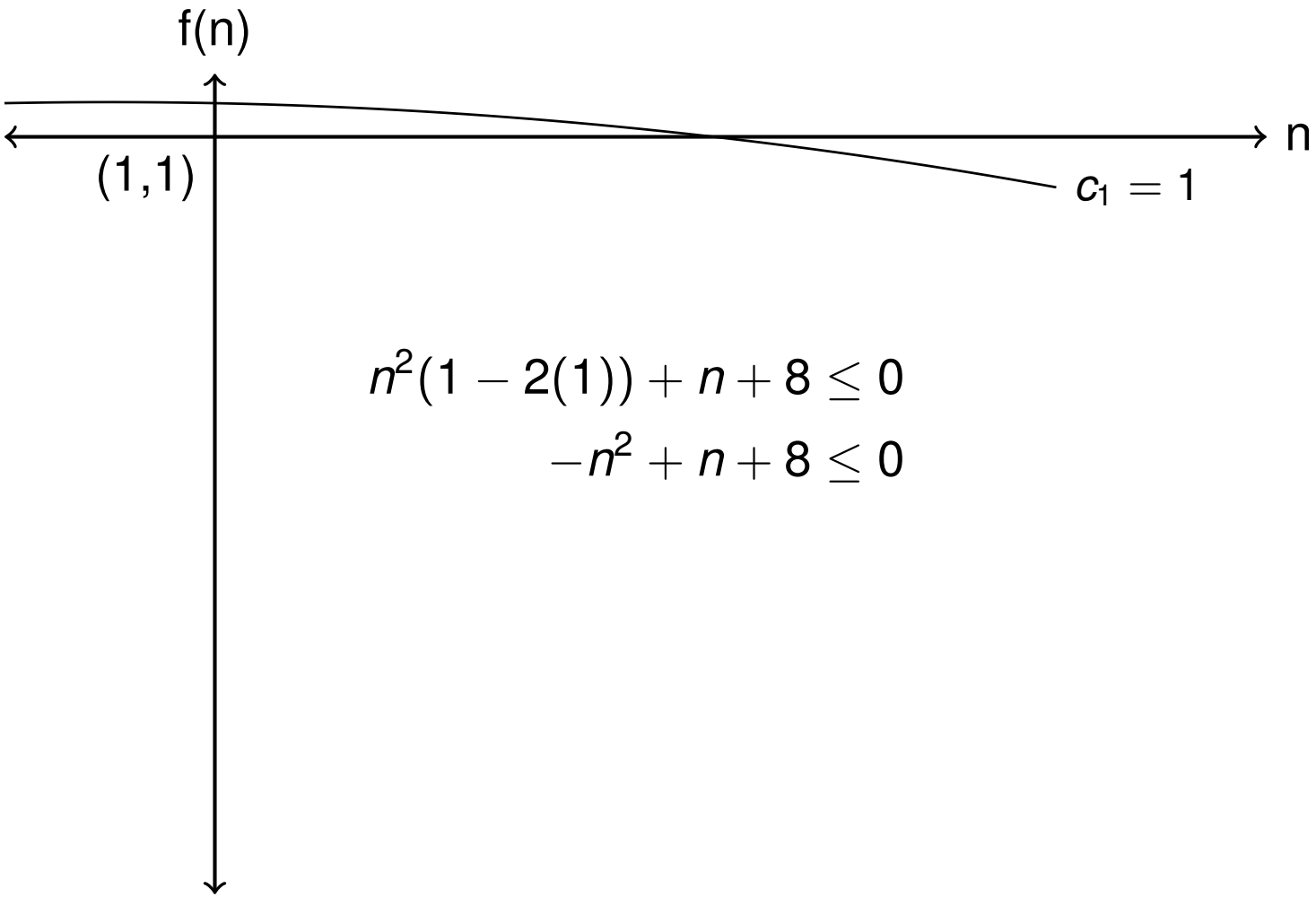
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



Análise de $n^2(1-2c_1)+n+8 \leq 0$

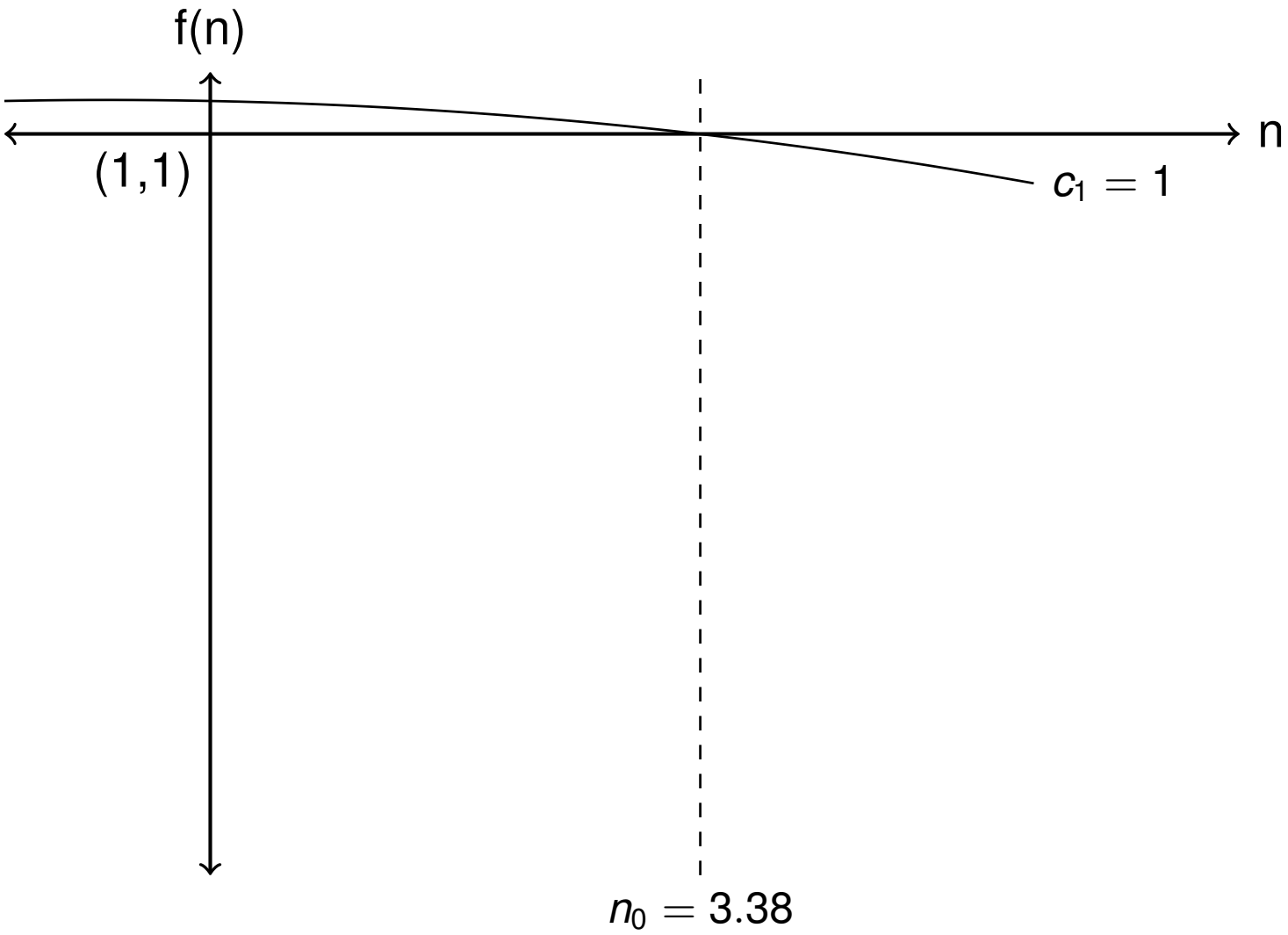
Escolha de c_1:



$$n^2(1-2(1))+n+8 \leq 0$$
$$-n^2+n+8 \leq 0$$

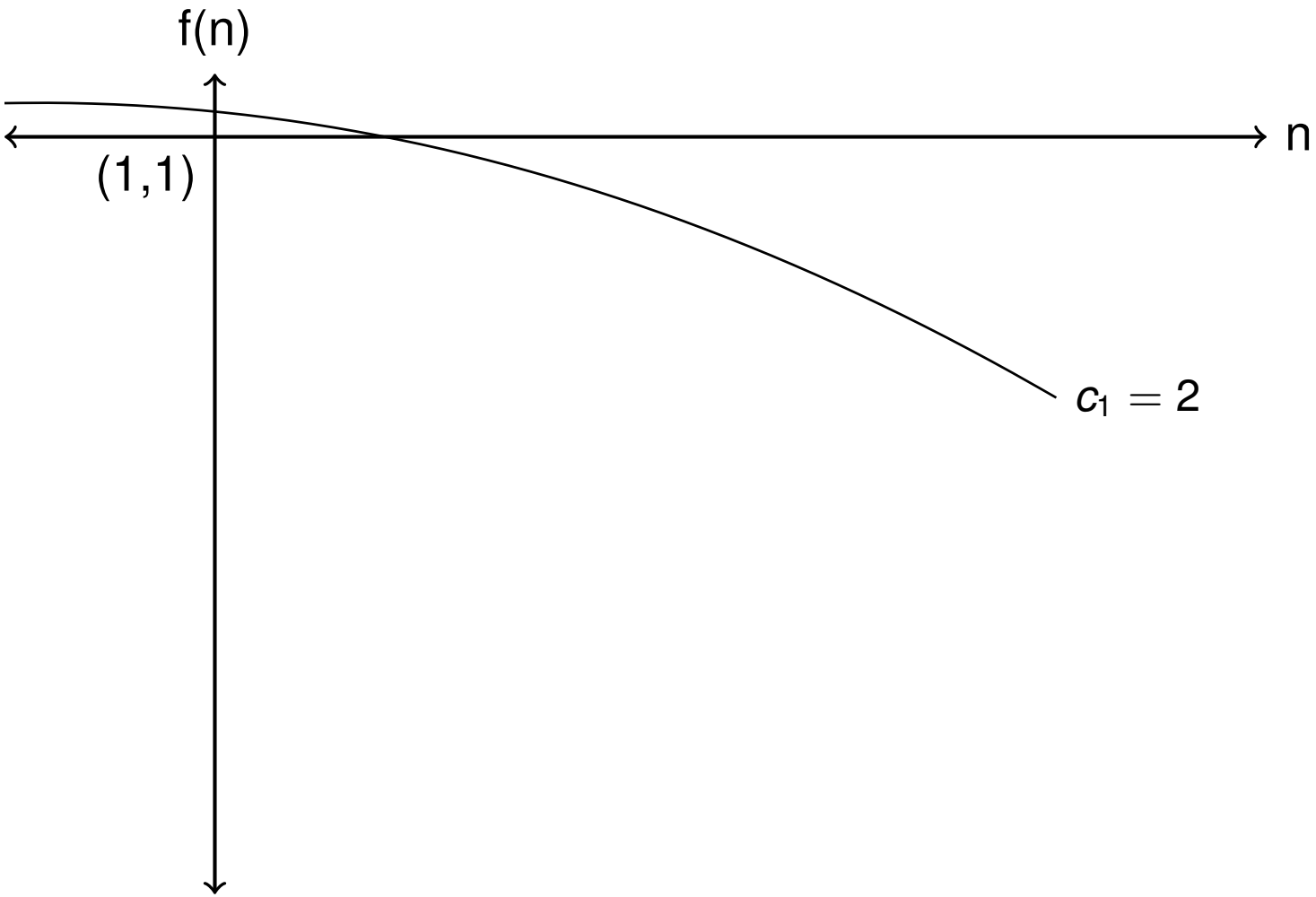
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



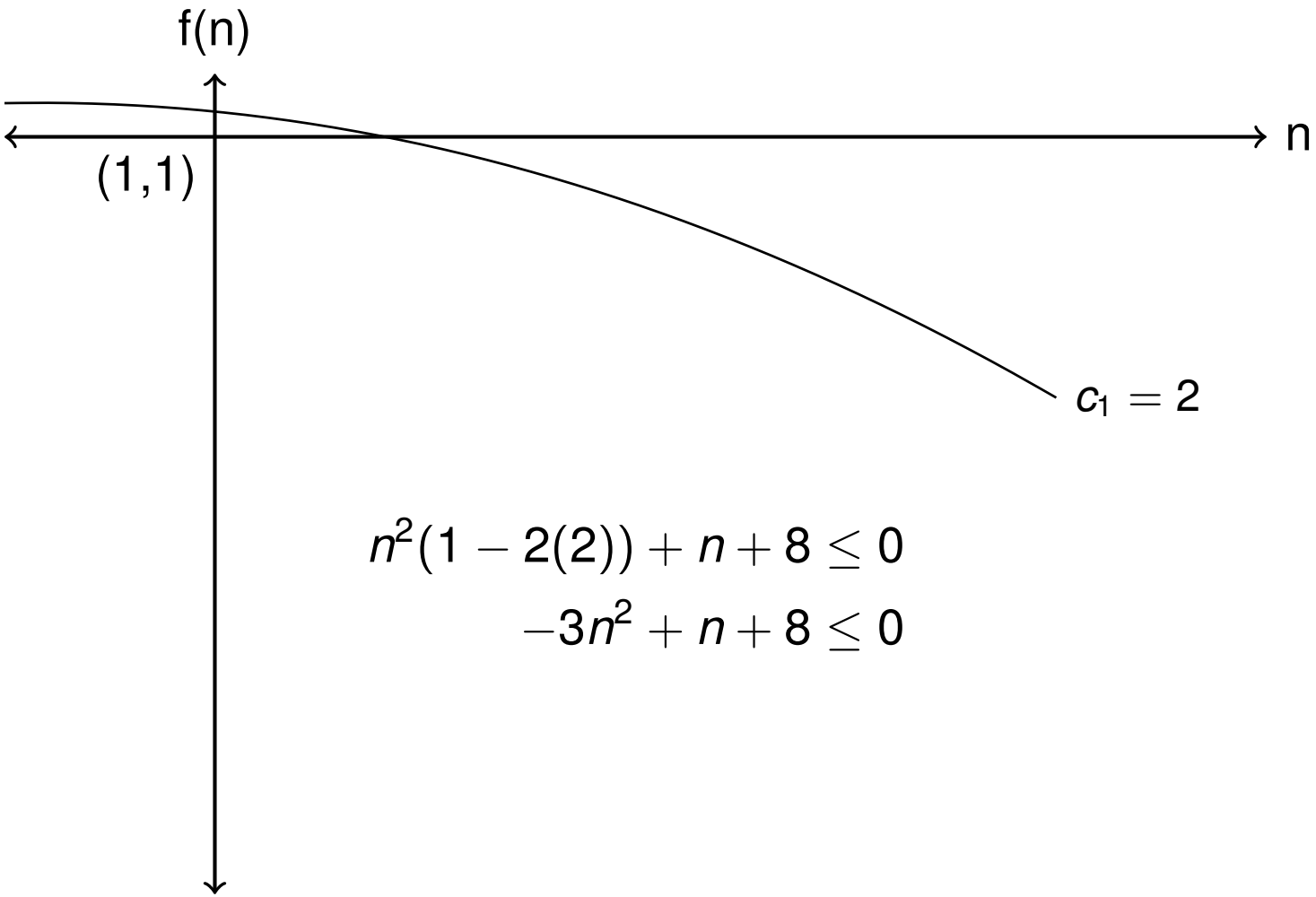
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



Análise de $n^2(1-2c_1)+n+8 \leq 0$

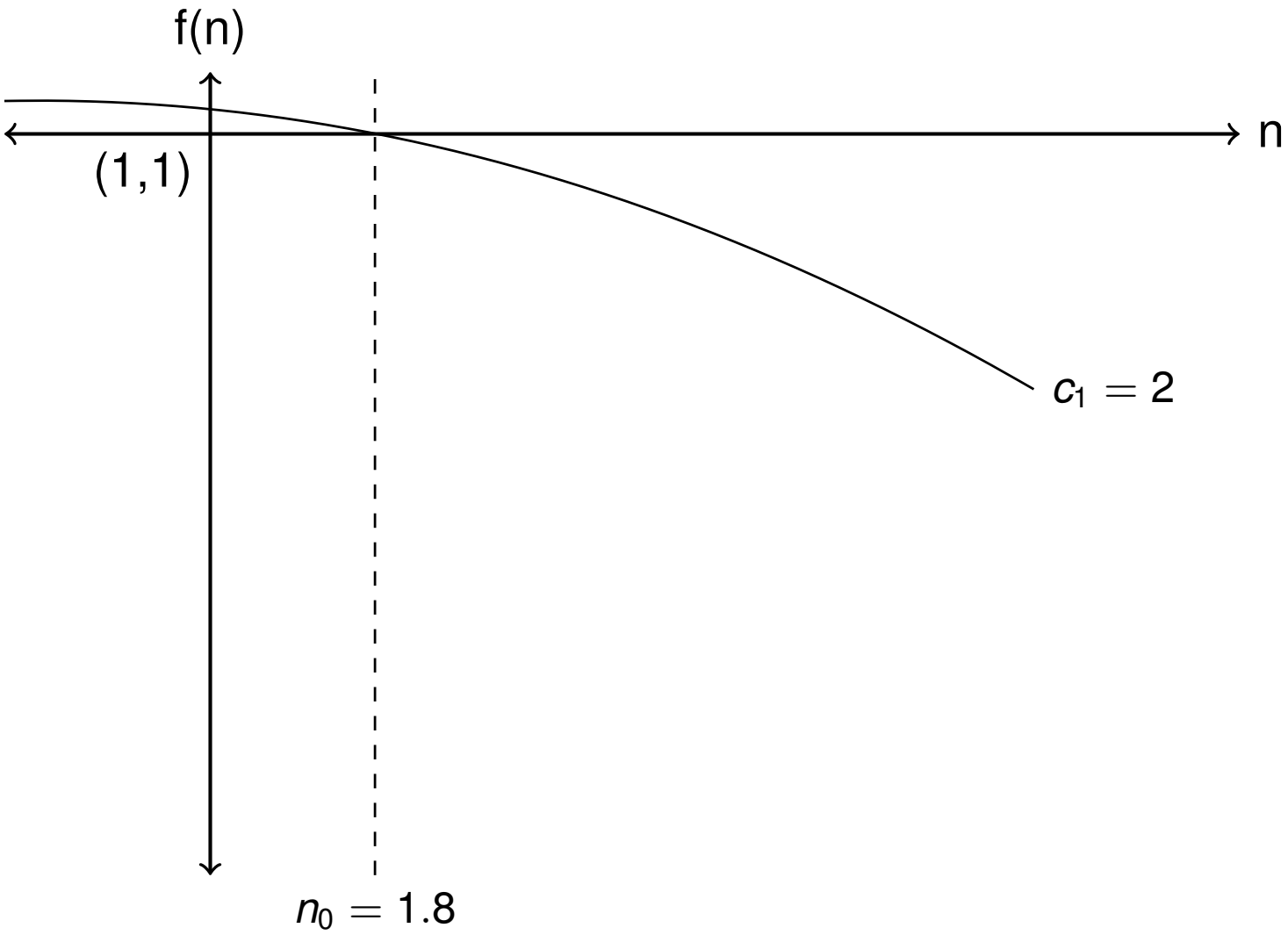
Escolha de c_1:



$$n^2(1-2(2))+n+8 \leq 0$$
$$-3n^2+n+8 \leq 0$$

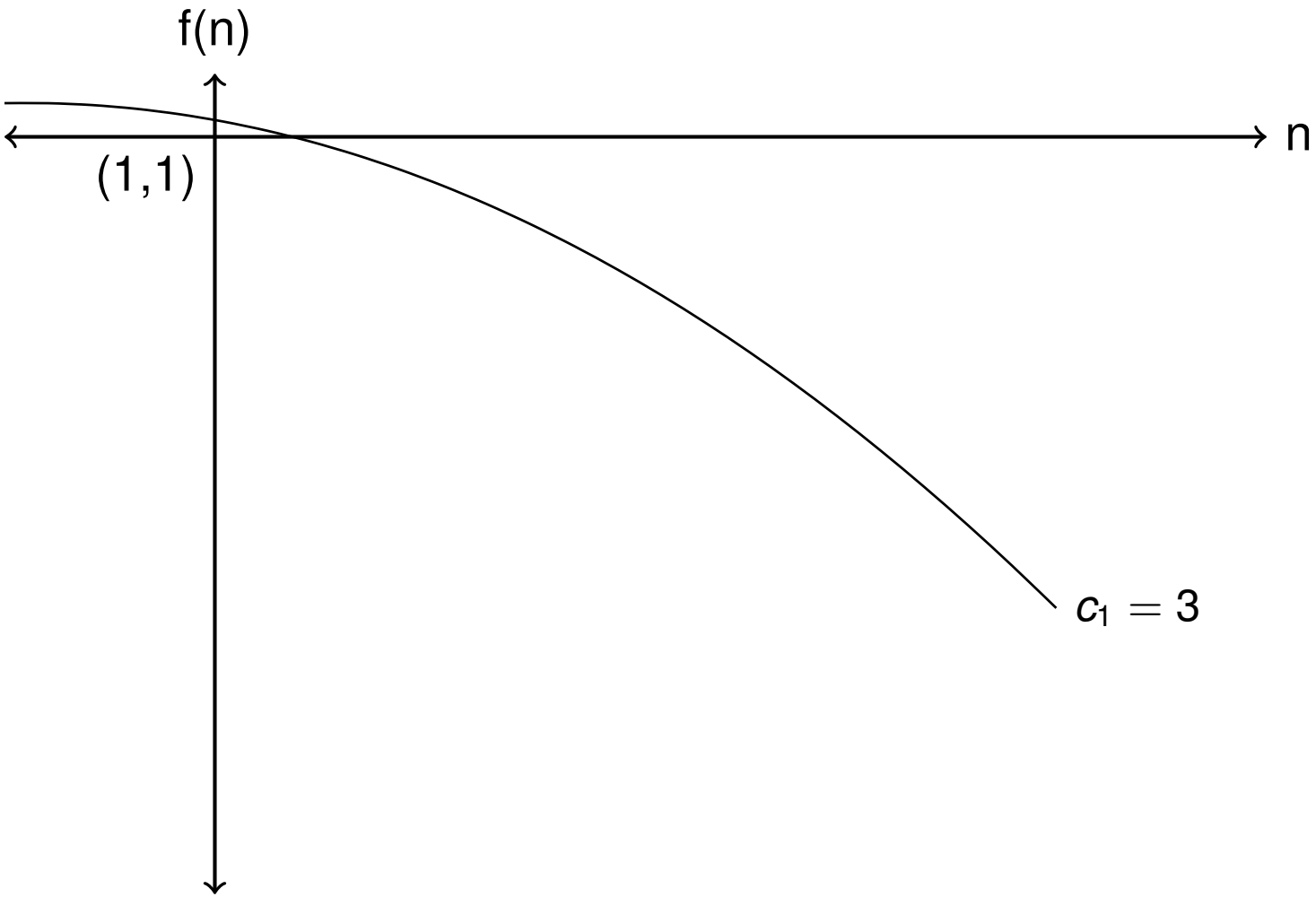
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



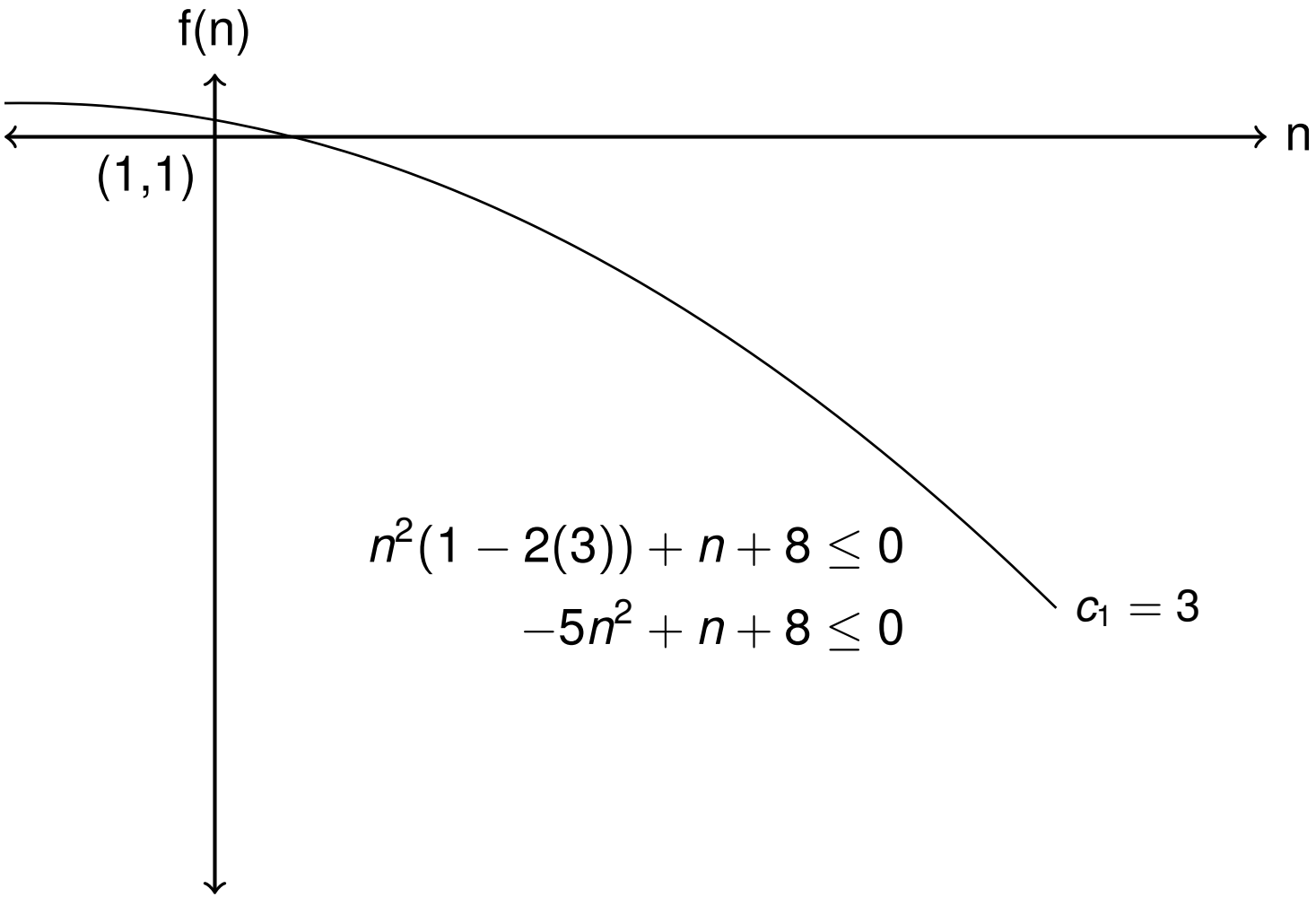
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



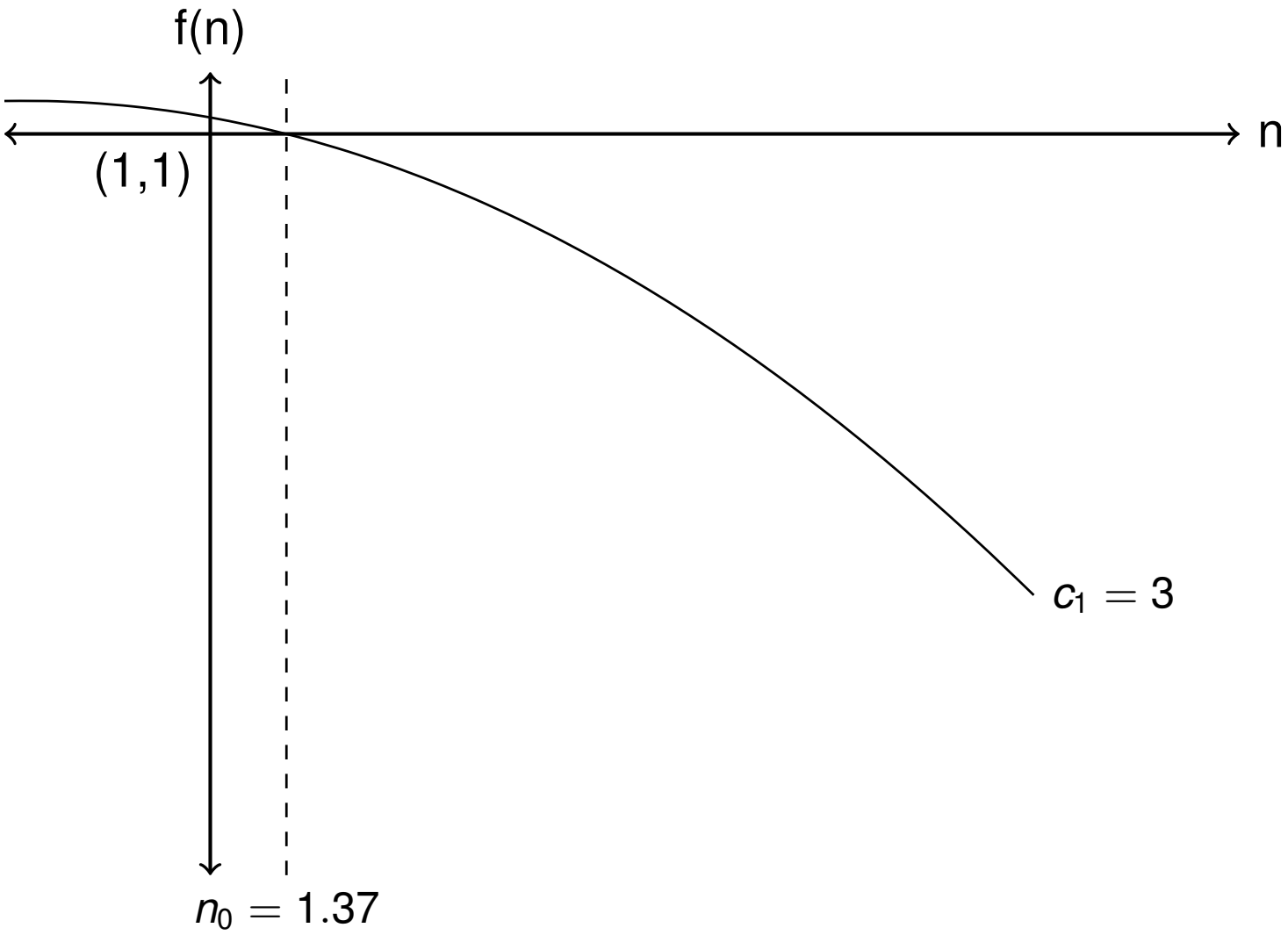
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



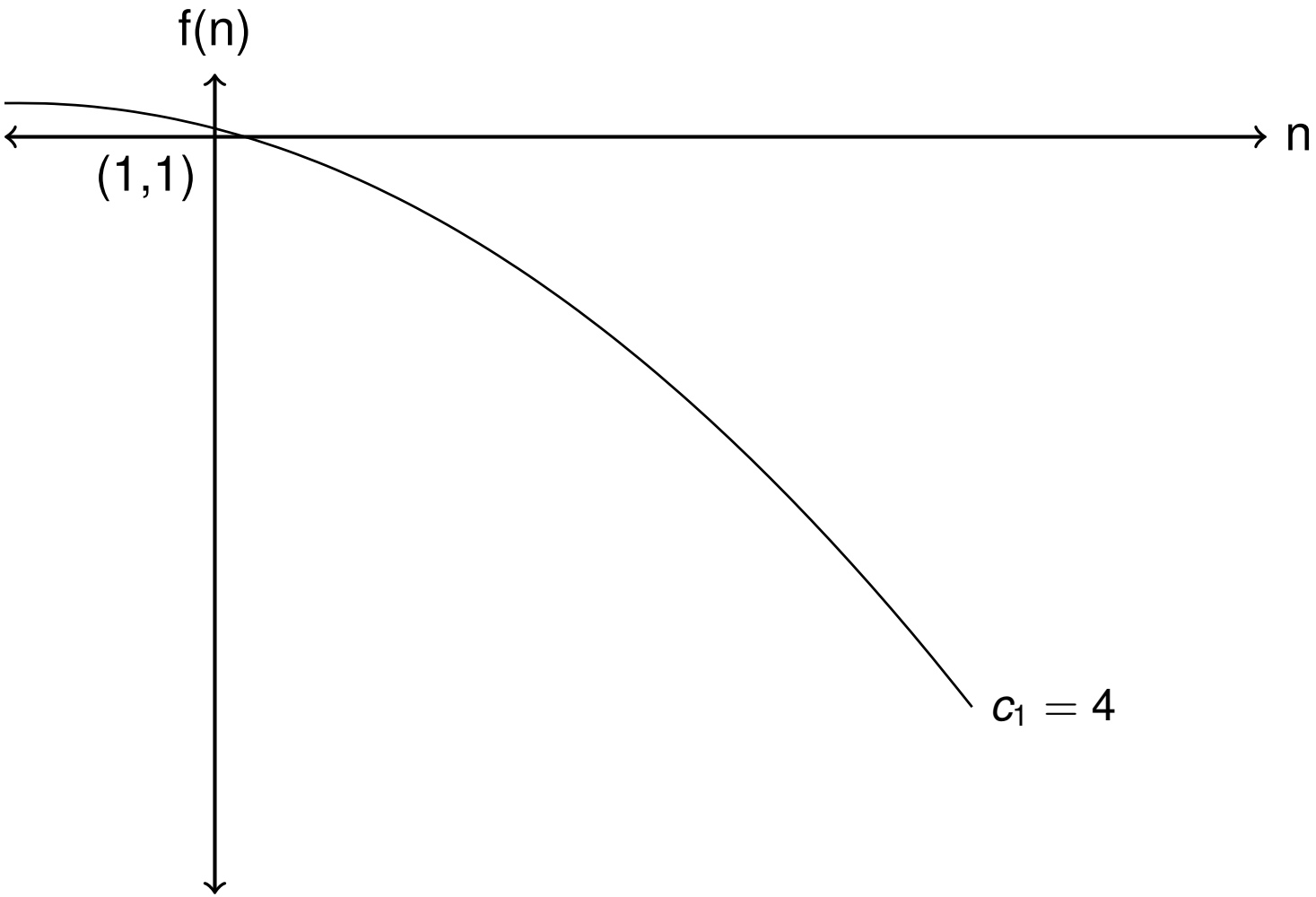
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



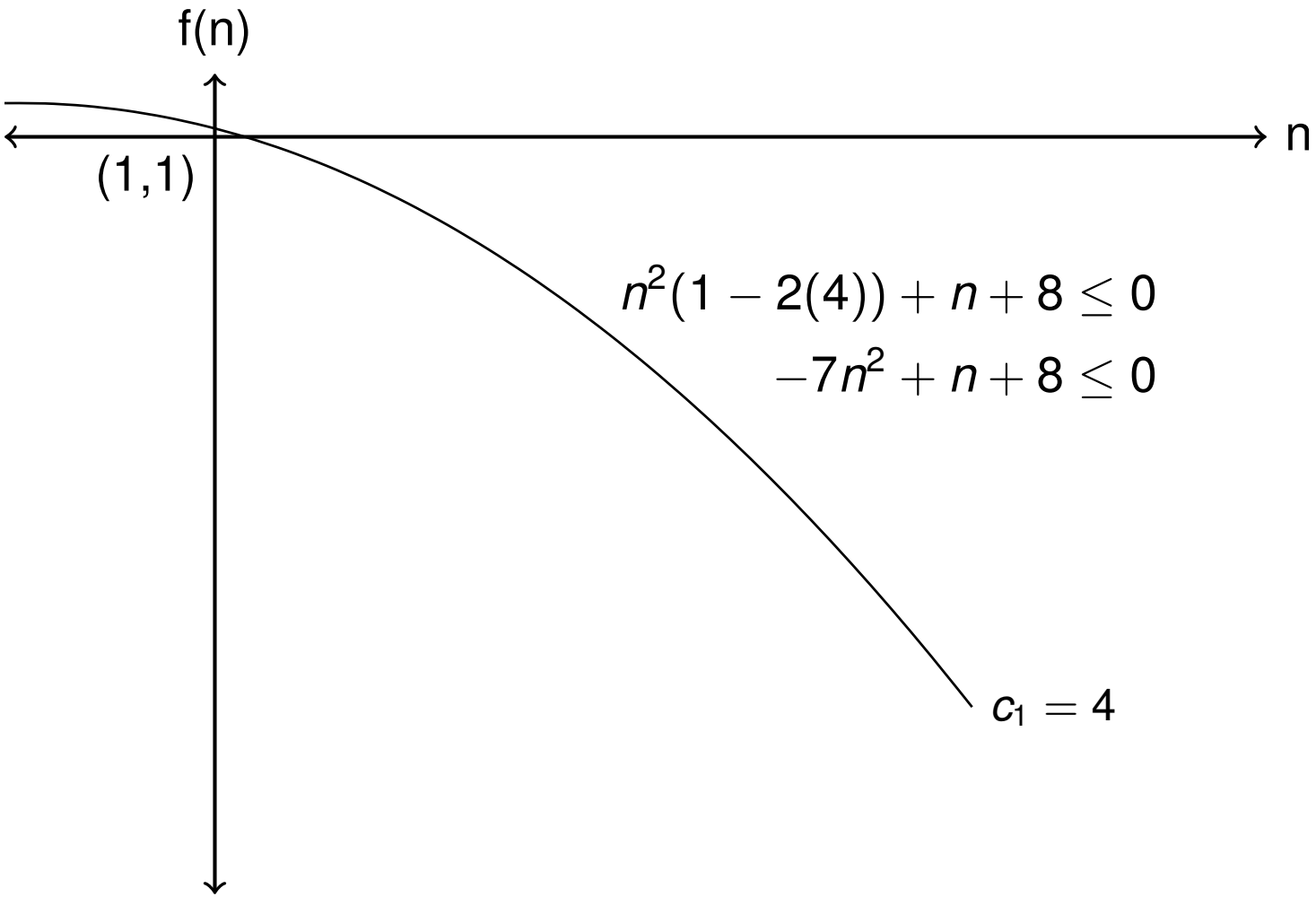
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



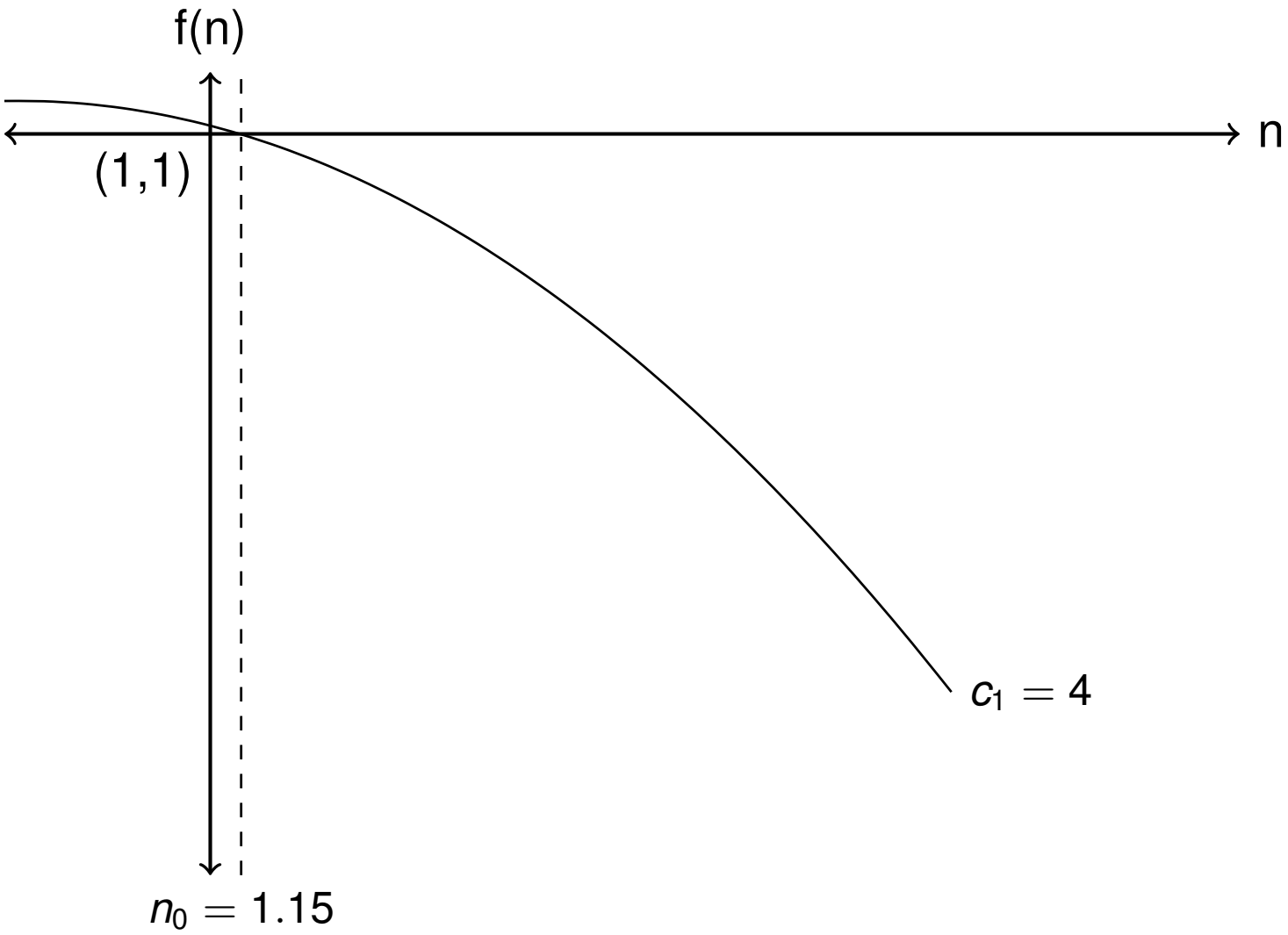
Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



Análise de $n^2(1-2c_1)+n+8 \leq 0$

Escolha de c_1:



Notação Ω

Uma função $f(n)$ é dita como sendo $\Omega(g(n))$ se existe uma constante positivas c para o qual:

$$0 < c \times g(n) \leq f(n)$$

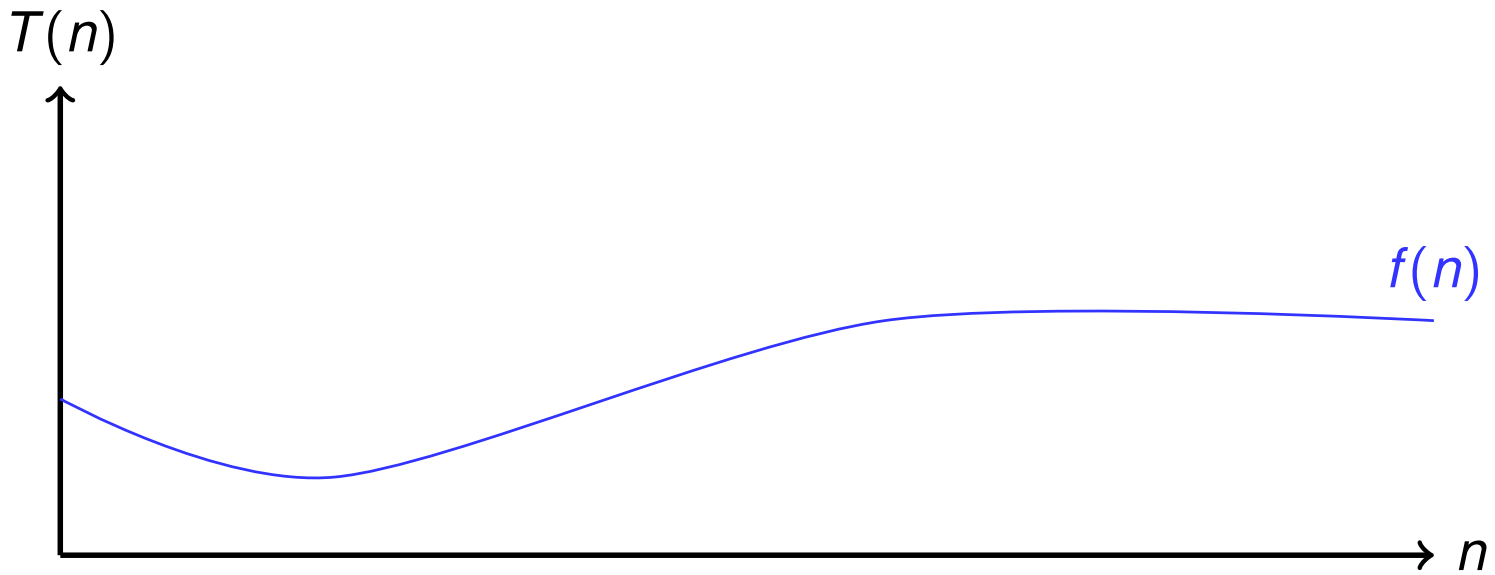
para todo $n > n_0$.

Notação Ω

Uma função $f(n)$ é dita como sendo $\Omega(g(n))$ se existe uma constante positivas c para o qual:

$$0 < c \times g(n) \leq f(n)$$

para todo $n > n_0$.

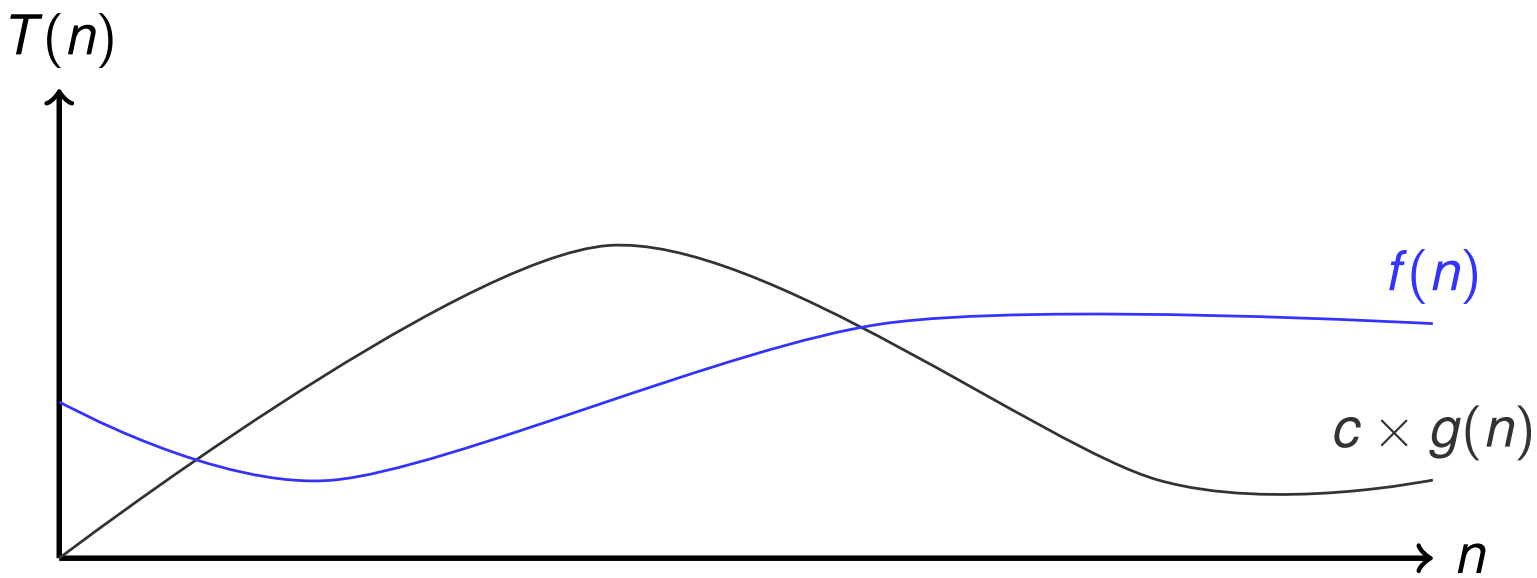


Notação Ω

Uma função $f(n)$ é dita como sendo $\Omega(g(n))$ se existe uma constante positivas c para o qual:

$$0 < c \times g(n) \leq f(n)$$

para todo $n > n_0$.

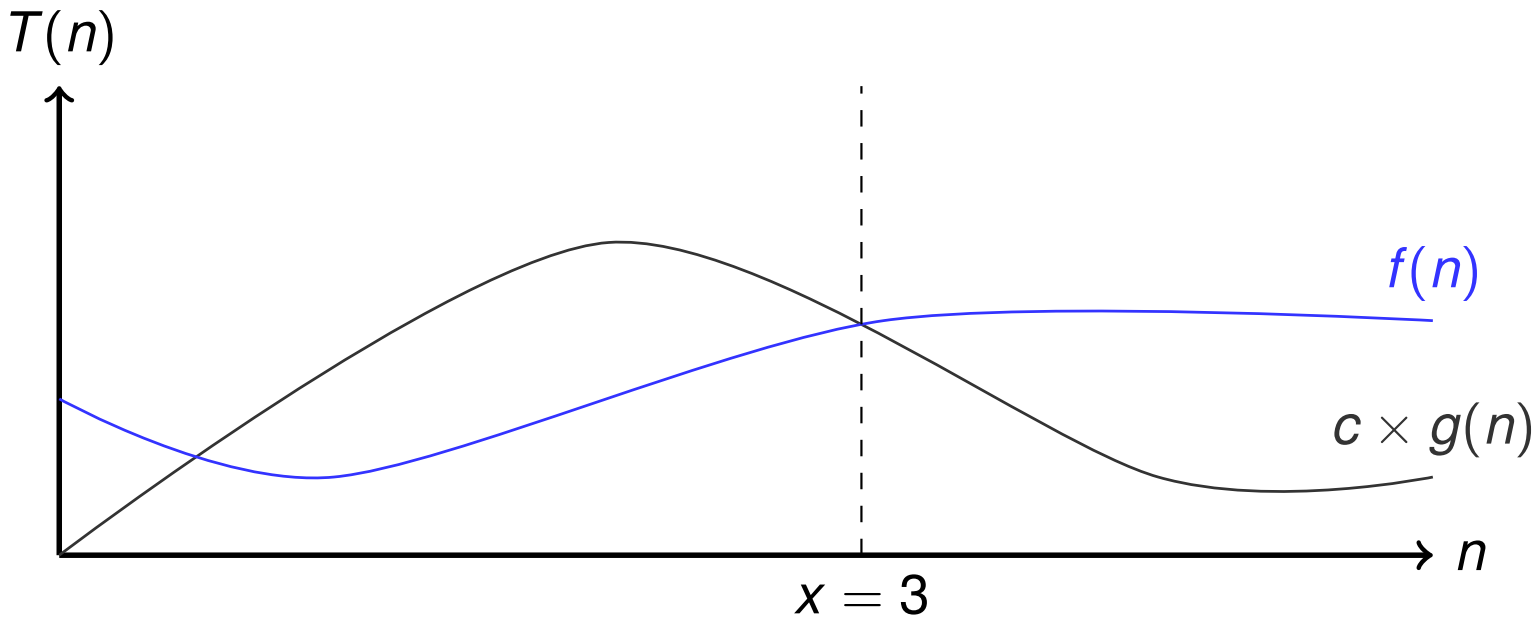


Notação Ω

Uma função $f(n)$ é dita como sendo $\Omega(g(n))$ se existe uma constante positivas c para o qual:

$$0 < c \times g(n) \leq f(n)$$

para todo $n > n_0$.

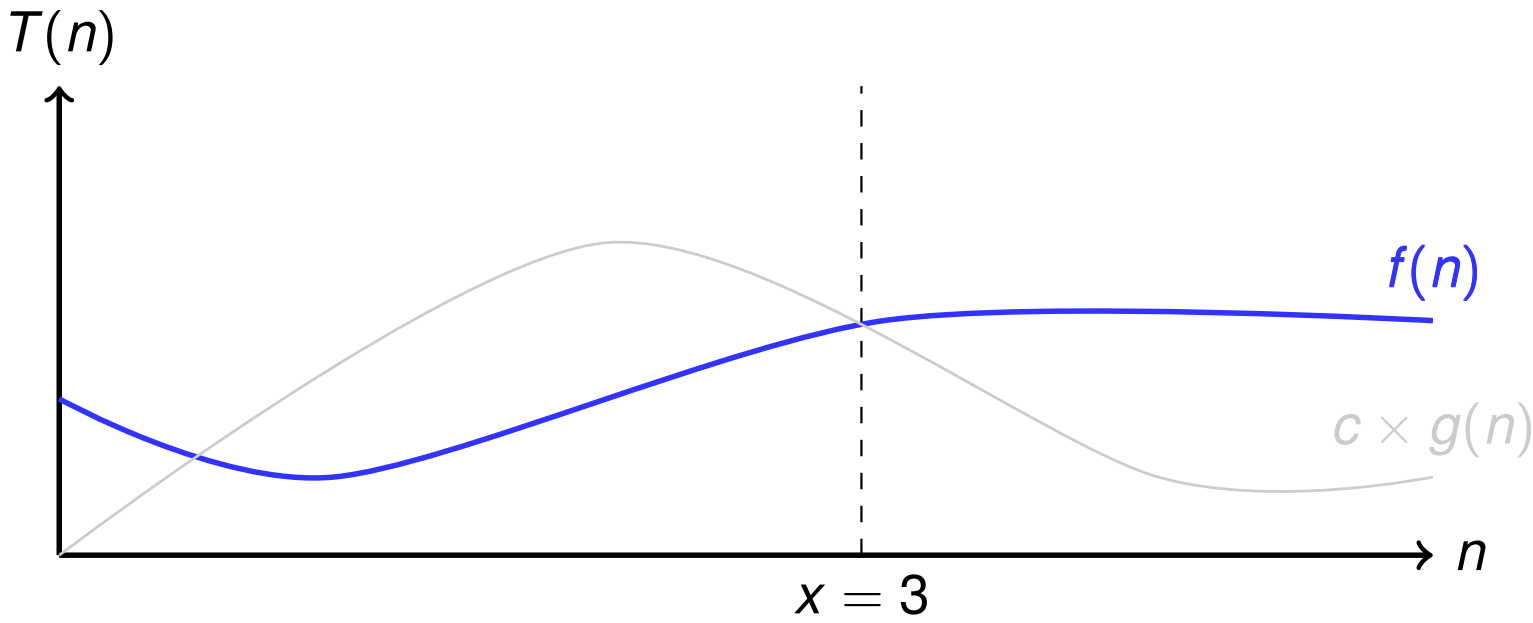


Notação Ω

Uma função $f(n)$ é dita como sendo $\Omega(g(n))$ se existe uma constante positivas c para o qual:

$$0 < c \times g(n) \leq f(n)$$

para todo $n > n_0$.



Exemplo: Complexidade Ω

Mostre que:

$$2n^2 - 20n - 50 \text{ é } \Omega(2n)$$

Resolvendo a inequação da notação Ω:

$$c_1 \times (2n) \leq 2n^2 - 20n - 50 \tag{1}$$

$$2n^2 - 20n - 50 - 2c_1 n \geq 0 \tag{2}$$

$$2n^2 - n(20 + 2c_1) - 50 \geq 0 \tag{3}$$

$$n^2 - n(10 + c_1) - 25 \geq 0 \tag{4}$$

portanto, precisamos avaliar a concavidade desta curva. Escolher um valor de c_1 no qual a inequação 4 seja válida para qualquer valor de $n \geq n_0$.

Exemplo: Complexidade Ω

Ao analisar a inequação:

$$n^2 - n(10 + c_1) - 25 \geq 0$$

Percebemos que se comporta como uma inequação do segundo grau no qual o produto e a soma das raízes são respectivamente $(10 + c_1)$ e (-25).
Graficamente:

Exemplo: Complexidade Ω

Ao analisar a inequação:

$$n^2 - n(10 + c_1) - 25 \geq 0$$

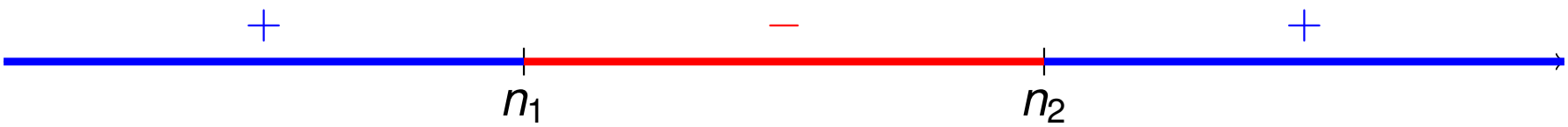
Percebemos que se comporta como uma inequação do segundo grau no qual o produto e a soma das raízes são respectivamente $(10 + c_1)$ e (-25).
Graficamente:

Exemplo: Complexidade Ω

Ao analisar a inequação:

$$n^2 - n(10 + c_1) - 25 \geq 0$$

Percebemos que se comporta como uma inequação do segundo grau no qual o produto e a soma das raízes são respectivamente $(10 + c_1)$ e (-25). Graficamente:



$$n_1 + n_2 = 10 + c_1$$
$$n_1 n_2 = -25$$

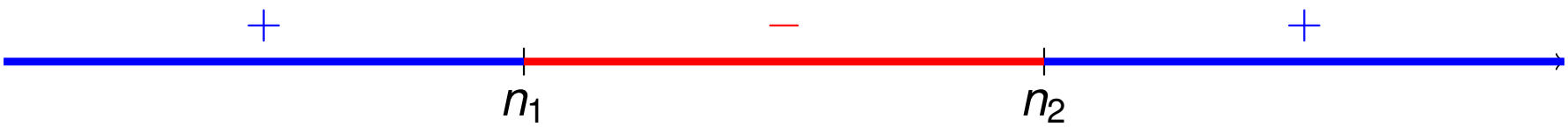
Escolhendo $c_1 = 2$, observamos que $n_1 \approx -13.81$ e $n_2 \approx -1.81$.

Exemplo: Complexidade Ω

Ao analisar a inequação:

$$n^2 - n(10 + c_1) - 25 \geq 0$$

Percebemos que se comporta como uma inequação do segundo grau no qual o produto e a soma das raízes são respectivamente $(10 + c_1)$ e (-25).
Graficamente:



$$n_1 + n_2 = 10 + c_1$$
$$n_1 n_2 = -25$$

Escolhendo $c_1 = 2$, observamos que $n_1 \approx -13.81$ e $n_2 \approx -1.81$.

Ou seja:
A partir de $n_0 = 2$, escolhendo $c_1 = 2$, a inequação será sempre válida.

Notação Θ

Uma função $f(n)$ é dita como sendo $\Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 para os quais:

$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$$

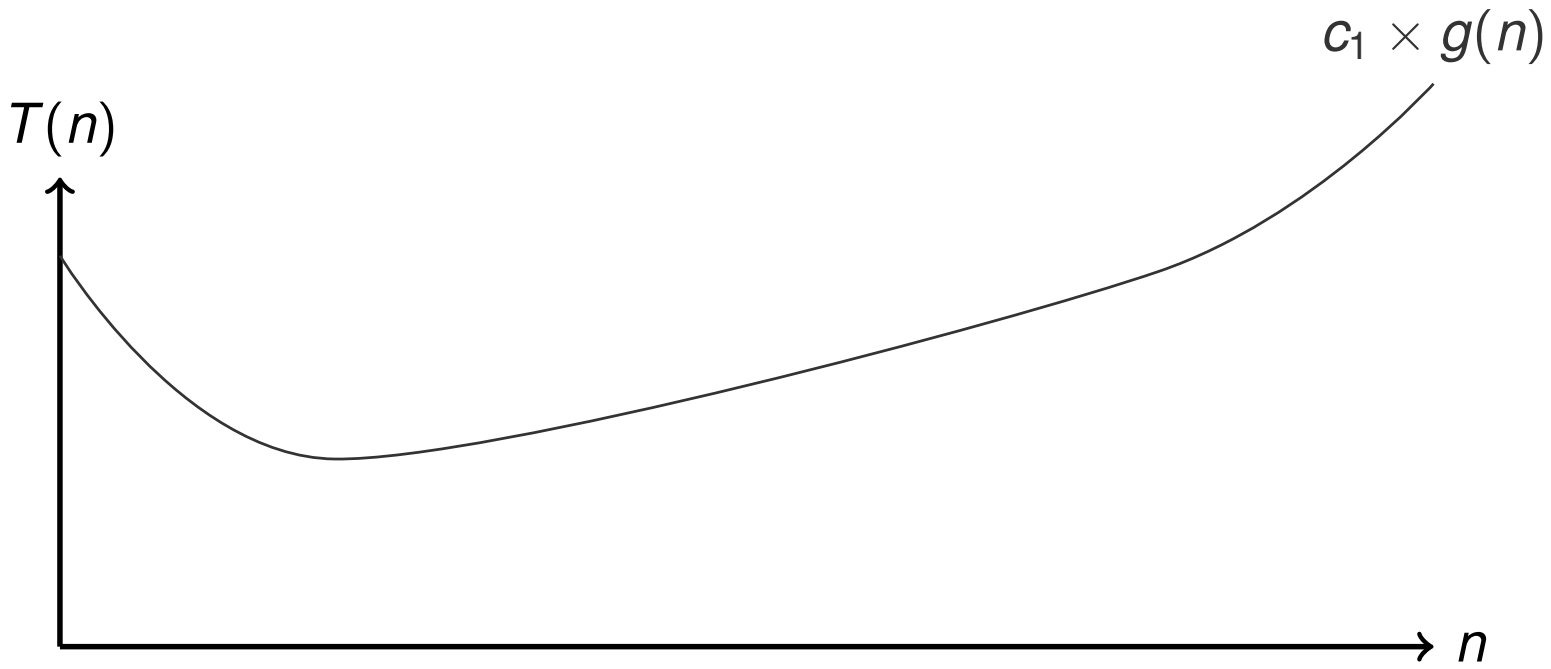
para todo $n > n_0$.

Notação Θ

Uma função $f(n)$ é dita como sendo $\Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 para os quais:

$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$$

para todo $n > n_0$.

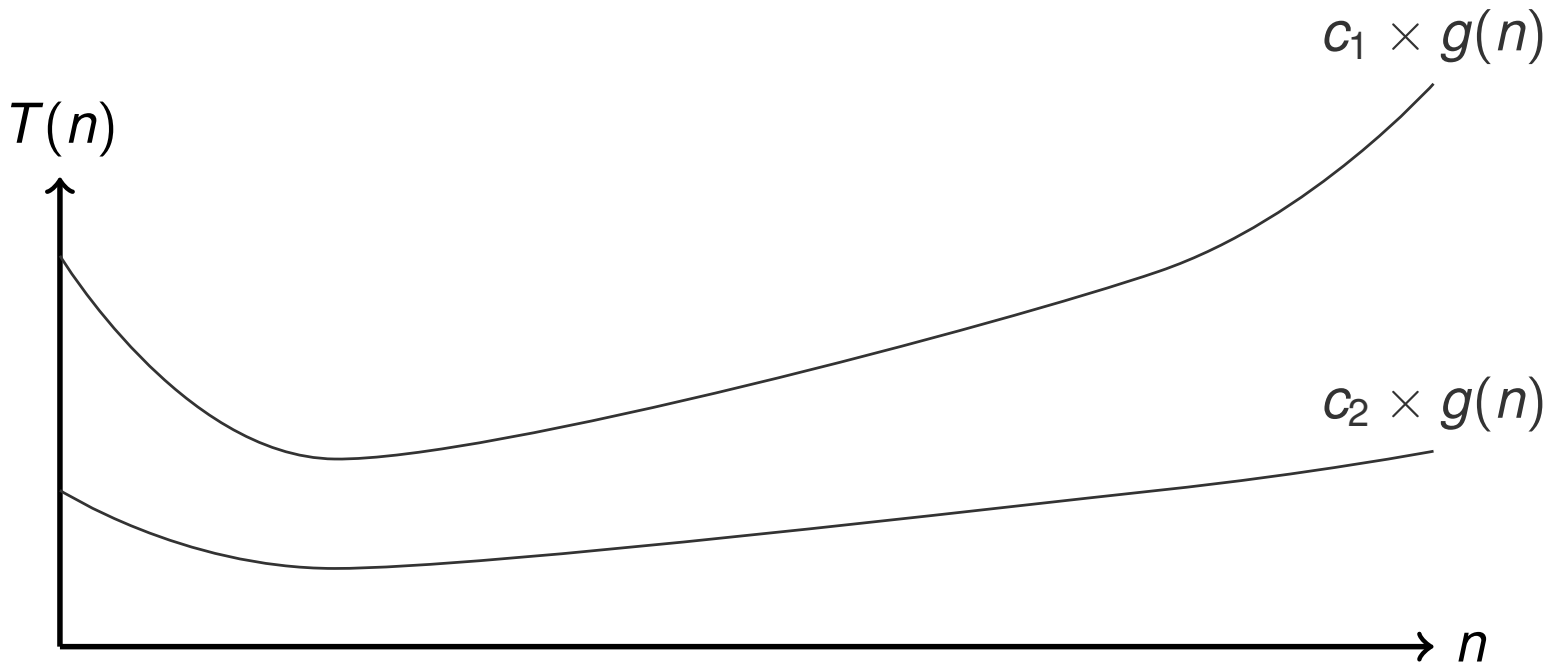


Notação Θ

Uma função $f(n)$ é dita como sendo $\Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 para os quais:

$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$$

para todo $n > n_0$.

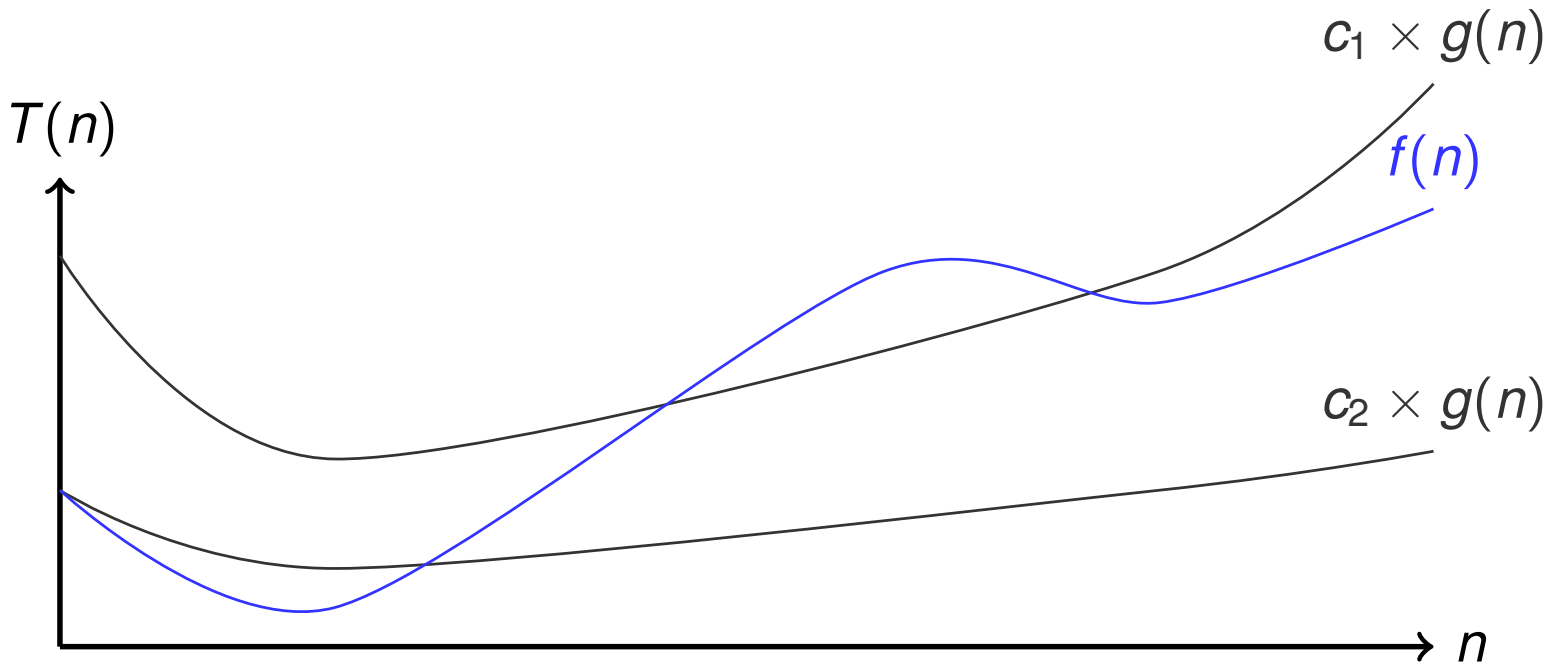


Notação Θ

Uma função $f(n)$ é dita como sendo $\Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 para os quais:

$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$$

para todo $n > n_0$.

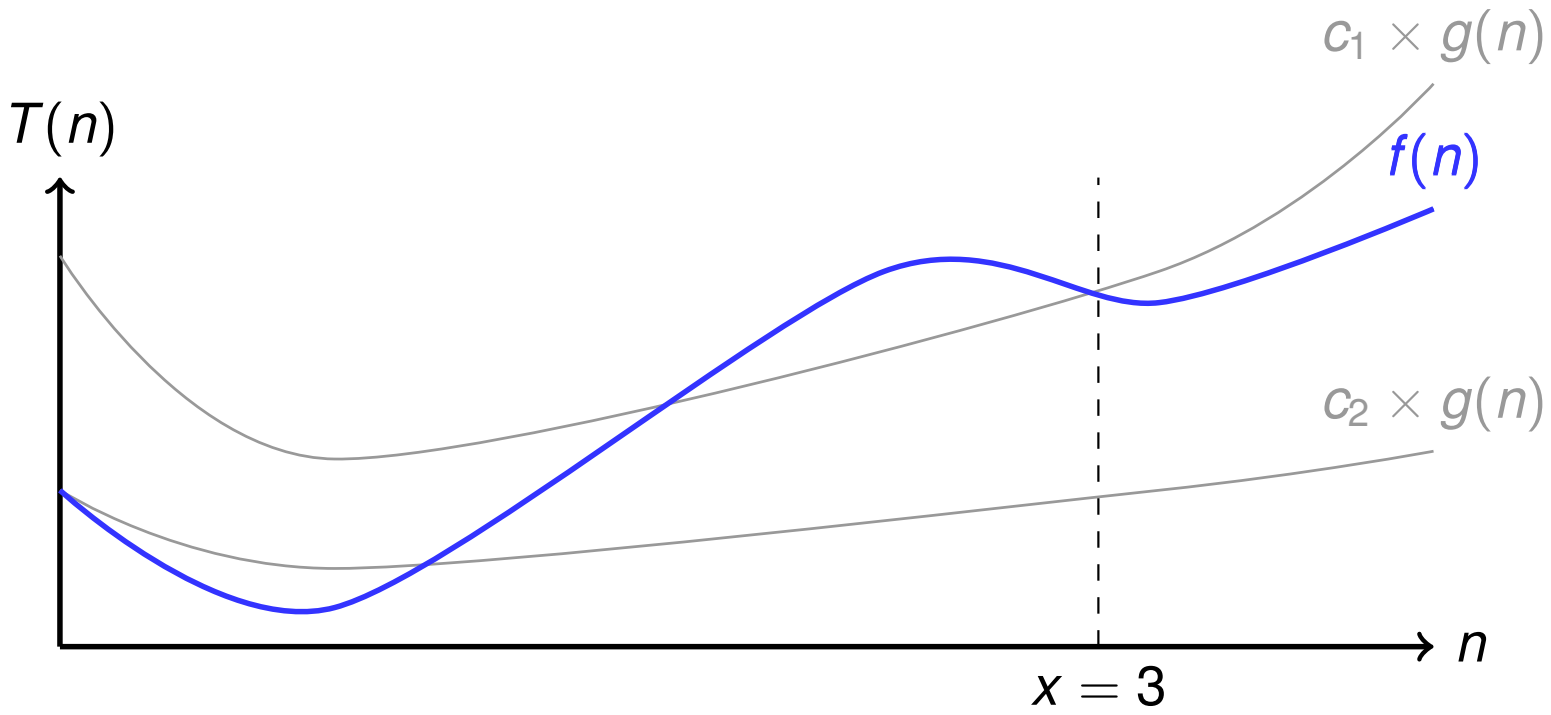


Notação Θ

Uma função $f(n)$ é dita como sendo $\Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 para os quais:

$$0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$$

para todo $n > n_0$.



Exemplo: Complexidade Θ

Verifique se:

$$\lg n \text{ é } \Theta(\log_{10} n)$$

Para ser $\Theta(\log_{10} n)$, a seguinte inequação deverá ser válida para um c_1 e c_2 positivos escolhidos arbitrariamente.

$$0 < c_1 \log_{10} n \leq \lg n \leq c_2 g(n) \log_{10}$$

Devemos então resolver separadamente estas inequações sabendo que:

$$\log_{10} n = \frac{\lg n}{\lg 10}$$

Exemplo: Complexidade $\ominus$

[1] Verifique esta afirmação

Exemplo: Complexidade $\ominus$

[1] Verifique esta afirmação

Exemplo: Complexidade Θ

Parte 1:

$$c_1 \log_{10} n \leq \lg n$$

$$c_1 \frac{\lg n}{\lg 10} \leq \lg n$$

$$c_1 \leq \lg 10$$

[1] Verifique esta afirmação

Exemplo: Complexidade Θ

Parte 1:

$$c_1 \log_{10} n \leq \lg n$$

$$c_1 \frac{\lg n}{\lg 10} \leq \lg n$$

$$c_1 \leq \lg 10$$

Parte 2:

$$\lg n \leq c_2 \log_{10} n$$

$$\lg n \leq c_2 \frac{\lg n}{\lg 10}$$

$$c_2 \geq \lg 10$$

[1] Verifique esta afirmação

Exemplo: Complexidade Θ

Parte 1:

$$c_1 \log_{10} n \leq \lg n$$

$$c_1 \frac{\lg n}{\lg 10} \leq \lg n$$

$$c_1 \leq \lg 10$$

Parte 2:

$$\lg n \leq c_2 \log_{10} n$$

$$\lg n \leq c_2 \frac{\lg n}{\lg 10}$$

$$c_2 \geq \lg 10$$

Ou seja, podemos escolher os valores:

$$c_1 = 2 \leq \lg 10$$

$$c_2 = 3 \geq \lg 10$$

[1] Verifique esta afirmação

Exemplo: Complexidade Θ

Parte 1:

$$c_1 \log_{10} n \leq \lg n$$

$$c_1 \frac{\lg n}{\lg 10} \leq \lg n$$

$$c_1 \leq \lg 10$$

Parte 2:

$$\lg n \leq c_2 \log_{10} n$$

$$\lg n \leq c_2 \frac{\lg n}{\lg 10}$$

$$c_2 \geq \lg 10$$

Ou seja, podemos escolher os valores:

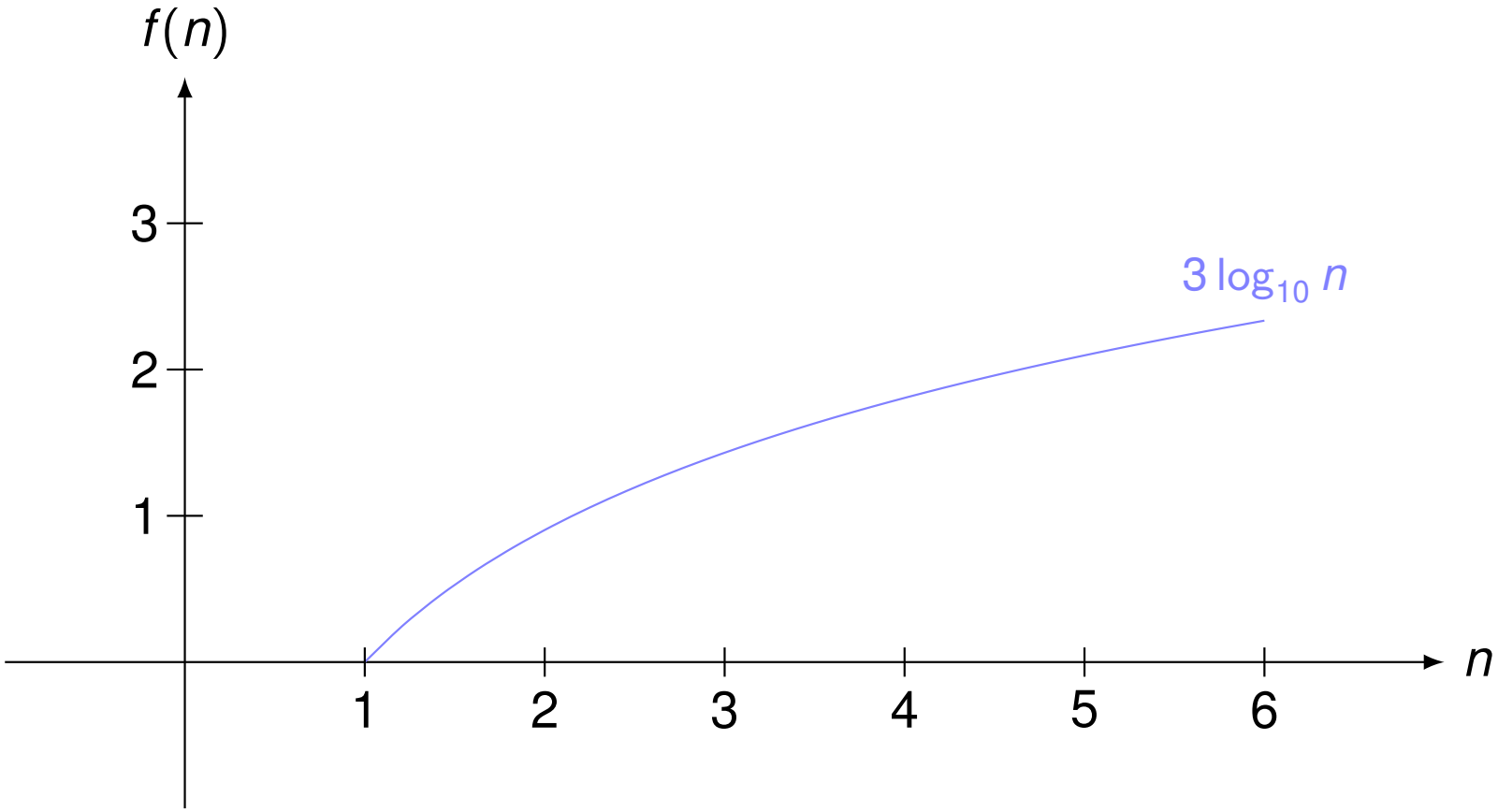
$$c_1 = 2 \leq \lg 10$$
$$c_2 = 3 \geq \lg 10$$

Deste modo, a partir de $n_0 > 0$[1]:

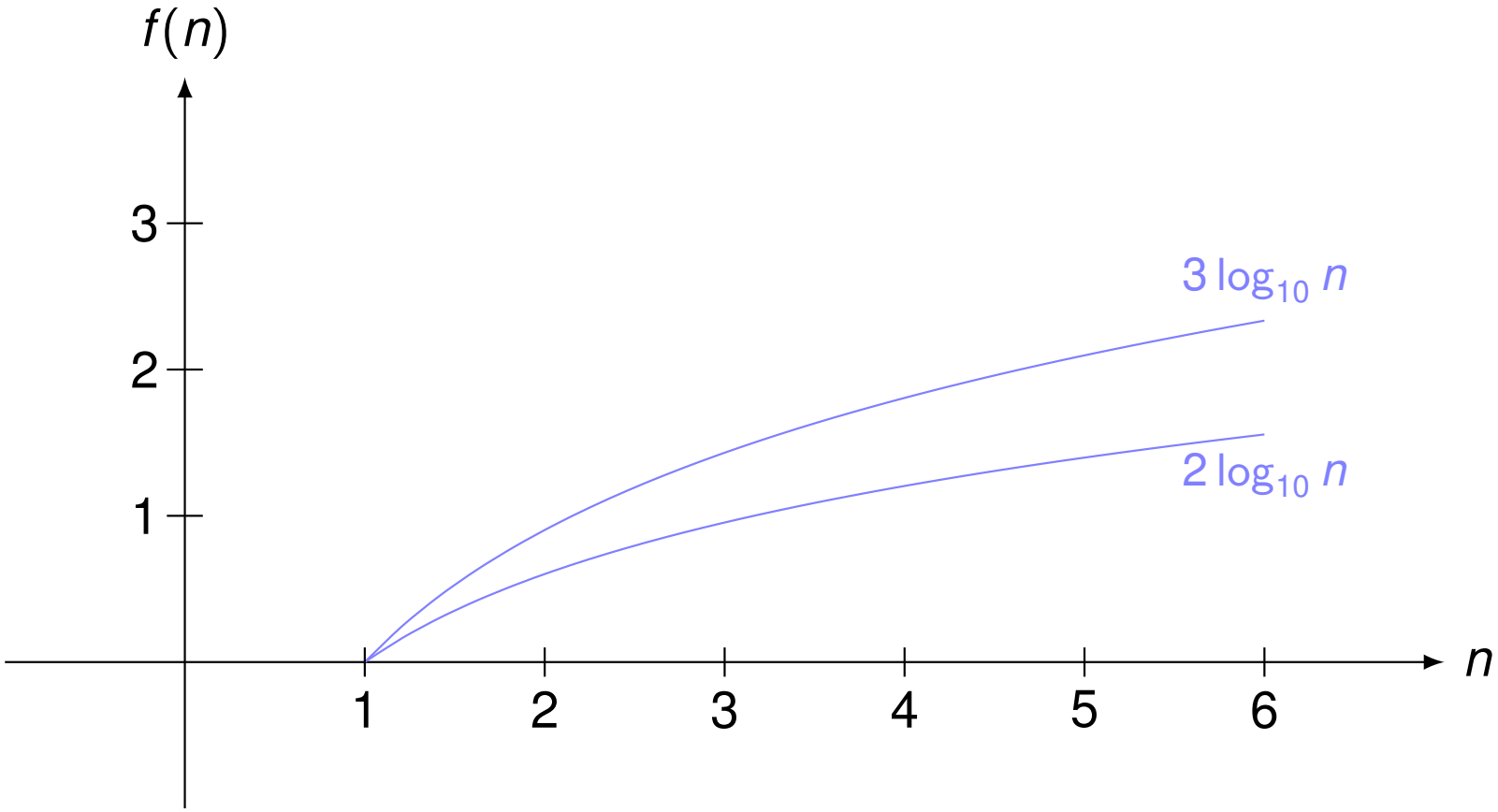
$$0 < c_1 \log_{10} n \leq \lg n \leq c_2 \log_{10} n$$

[1] Verifique esta afirmação

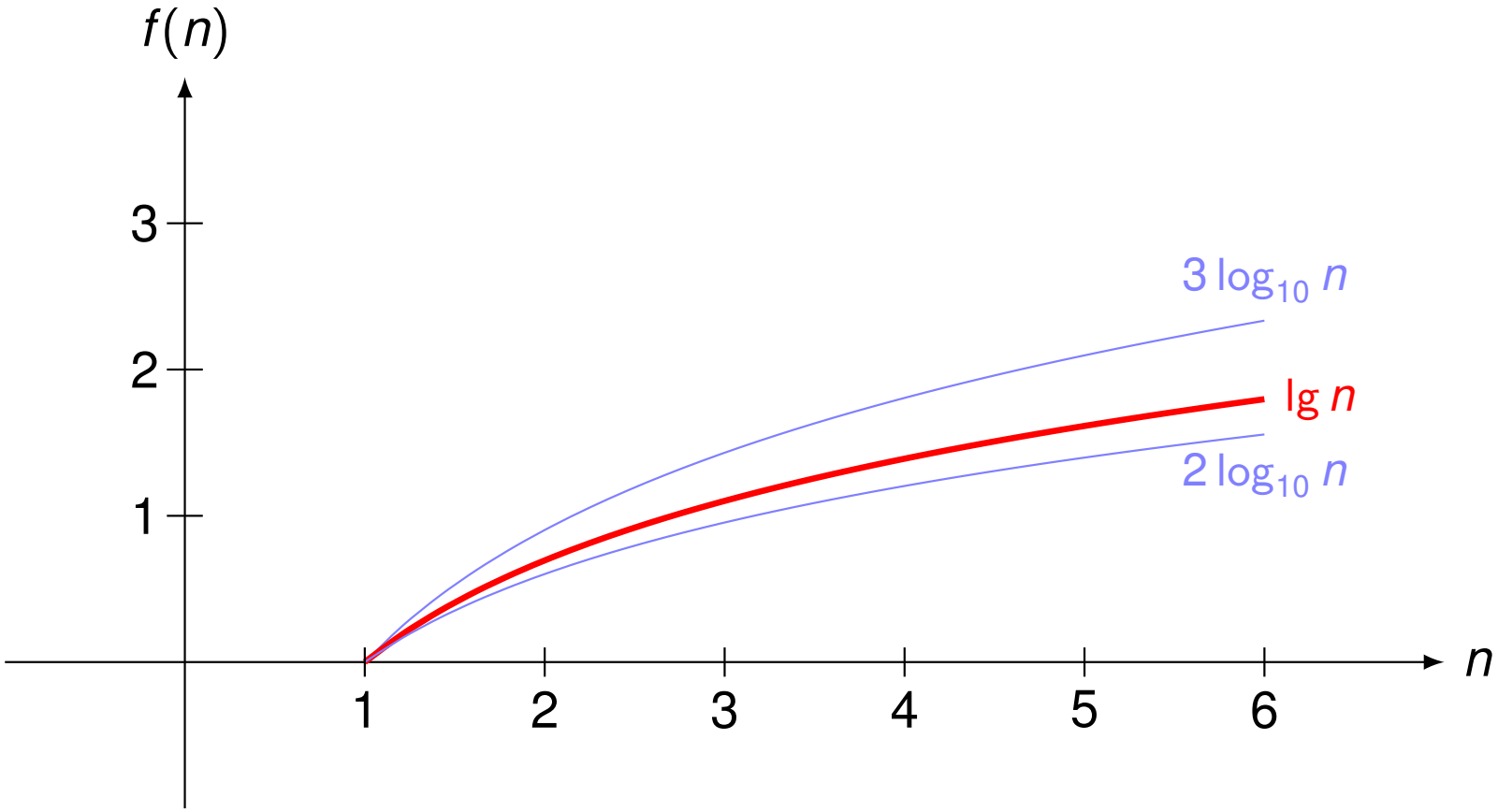
Exemplo: Complexidade Θ



Exemplo: Complexidade Θ



Exemplo: Complexidade Θ



Ordem de Complexidade

Complexidade	Descrição simplificada

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica
$O(n)$	Complexidade Linear

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica
$O(n)$	Complexidade Linear
$O(n\ log(n))$	Complexidade Log Linear

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica
$O(n)$	Complexidade Linear
$O(n\ log(n))$	Complexidade Log Linear
$O(n^2)$	Complexidade Quadrática

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica
$O(n)$	Complexidade Linear
$O(n\ log(n))$	Complexidade Log Linear
$O(n^2)$	Complexidade Quadrática
$O(n^3)$	Complexidade Cúbica

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica
$O(n)$	Complexidade Linear
$O(n\ log(n))$	Complexidade Log Linear
$O(n^2)$	Complexidade Quadrática
$O(n^3)$	Complexidade Cúbica
$O(2^n)$	Complexidade Exponencial

Ordem de Complexidade

Complexidade	Descrição simplificada
$O(1)$	Complexidade Constante
$O(log(n))$	Complexidade logarítmica
$O(n)$	Complexidade Linear
$O(n\ log(n))$	Complexidade Log Linear
$O(n^2)$	Complexidade Quadrática
$O(n^3)$	Complexidade Cúbica
$O(2^n)$	Complexidade Exponencial
$O(n!)$	Complexidade Fatorial

Complexidade Constante

Complexidade Constante

- Ocorre quando a complexidade do algoritmo independe do tamanho do problema (n);

Complexidade Constante

- Ocorre quando a complexidade do algoritmo independe do tamanho do problema (n);
- As instruções são executadas em uma quantidade fixa de vezes.

Complexidade Constante

- Ocorre quando a complexidade do algoritmo independe do tamanho do problema (n);
- As instruções são executadas em uma quantidade fixa de vezes.

Exemplo:

```
1  int funcConstante(int *v, int n){
2    if(V[0] > V[n-1]){
3      return V[n-1];
4    }else{
5      return V[0];
6    }
7  }
```

Complexidade Logarítmica

Complexidade Logarítmica

- Ocorre quando um problema é subdividido em em subproblemas menores.

Complexidade Logarítmica

- Ocorre quando um problema é subdividido em em subproblemas menores.
- A solução deve ser apenas uma das soluções menores.

Complexidade Logarítmica

- Ocorre quando um problema é subdividido em em subproblemas menores.
- A solução deve ser apenas uma das soluções menores.

Exemplo:

```
1   bool funcLog(int v[], int a, int b, int x)
2   {
3       if (a > b)
4           return false;
5       int c = (a + b) / 2;
6       if (v[c] == x)
7           return true;
8       else if (v[c] > x)
9           return funcLog(v, a, c-1, x);
10      else
11          return funcLog(v, c+1, b, x);
12  }
```

Complexidade Linear

Complexidade Linear

- Ocorre quando a complexidade do algoritmo é proporcional ao tamanho do problema;

Complexidade Linear

- Ocorre quando a complexidade do algoritmo é proporcional ao tamanho do problema;
- O tempo computacional é um múltiplo de n podendo ser acrescido de alguma constante.

Complexidade Linear

- Ocorre quando a complexidade do algoritmo é proporcional ao tamanho do problema;
- O tempo computacional é um múltiplo de *n* podendo ser acrescido de alguma constante.

Exemplo:

```
1   int funcLinear(int *v, int n){
2     int maior = v[0];
3     int cont = 0;
4     while(cont < n){
5       if(v[cont] > maior){
6           maior = v[i];
7       }
8       cont = cont + 1;
9     }
10    return maior;
11  }
```

Complexidade LogLinear

Complexidade LogLinear

- Ocorre quando o problema é subdividido em subproblemas;

Complexidade LogLinear

- Ocorre quando o problema é subdividido em subproblemas;
- A solução será a união das contribuiçoes das resoluções dos subproblemas.

Complexidade LogLinear

- Ocorre quando o problema é subdividido em subproblemas;
- A solução será a união das contribuiçoes das resoluções dos subproblemas.

Exemplo:

```
1  void funcLogLinear (int inicio, int fim){
2      if ( inicio < fim ){
3          int meio = ( inicio + fim )/2;
4          funcLogLinear ( inicio , meio );
5          funcLogLinear ( meio +1 , fim );
6          unir ( inicio , meio , fim );
7      }
8  }
```

Complexidade Quadrática

Complexidade Quadrática

- Geralmente está associado a **dois laços** que são proporcionais ao tamanho do problema.

Complexidade Quadrática

- Geralmente está associado a **dois laços** que são proporcionais ao tamanho do problema.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade quadrática.

Complexidade Quadrática

- Geralmente está associado a **dois laços** que são proporcionais ao tamanho do problema.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade quadrática.

Exemplo:

```
1   void funcQuadratica (int *v, int n){
2       int aux;
3       for(int i=0; i<n; i++)
4           for(int j=i+1; j<n; j++)
5               if(v[i] > V[j]){
6                   aux = v[i];
7                   v[i] = v[j];
8                   v[j] = aux;
9               }
10  }
```

Complexidade Cúbica

Complexidade Cúbica

- Geralmente está associado a **três laços** que são proporcionais ao tamanho do problema.

Complexidade Cúbica

- Geralmente está associado a **três laços** que são proporcionais ao tamanho do problema.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade Cúbica.

Complexidade Cúbica

- Geralmente está associado a **três laços** que são proporcionais ao tamanho do problema.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade Cúbica.

Exemplo:

```
1  void funcCubica (int *v, int n){
2      int aux;
3      for(int i=0; i<n; i++)
4          for(int j=i+1; j<n; j++)
5              for(int k=j+1; j<n; j++)
6                  aux = v[i] + v[j] + v[k];
7                  printf("%d", aux);
8  }
```

Complexidade Exponencial (2^n)

Complexidade Exponencial (2^n)

- Geralmente está associado a um problema que a resolução é feita por **força-bruta**.

Complexidade Exponencial (2^n)

- Geralmente está associado a um problema que a resolução é feita por **força-bruta**.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade Exponencial.

Complexidade Exponencial (2^n)

- Geralmente está associado a um problema que a resolução é feita por **força-bruta**.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade Exponencial.

Exemplo:

```
1  int funcExp(int n){
2      int aux;
3      if(n < 2)
4          return n;
5      else
6          return funcExp(n-1) + funcExp(n-2);
7   }
```

Complexidade Exponencial (2^n)

- Geralmente está associado a um problema que a resolução é feita por **força-bruta**.
- Qualquer uma outra complexidade anterior pode ser adicionada sem mudança na complexidade Exponencial.

Exemplo:

```
1  int funcExp(int n){
2      int aux;
3      if(n < 2)
4          return n;
5      else
6          return funcExp(n-1) + funcExp(n-2);
7   }
```

Exercício: Prove se é verdadeiro ou falso:

$$f(n) = 3^n \text{ é } O(2^n)$$

Complexidade Fatorial ($n!$)

Complexidade Fatorial ($n!$)

- Complexidade para algoritmo de força-bruta;

Complexidade Fatorial ($n!$)

- Complexidade para algoritmo de força-bruta;
- Problemas como caixeiro-viajante e criptografia podem se valer desta complexidade.

Pior, Melhor e Caso médio

Considere p algoritmo abaixo:

```
1  #define N 100000
2  int busca(int *A, int v){
3      for(int x = 0; x<N; x++){
4          if(A[x] == v)
5              return x;
6      }
7      return -1;
8  }
```

Pior, Melhor e Caso médio

Considere p algoritmo abaixo:

```
1  #define N 100000
2  int busca(int *A, int v){
3      for(int x = 0; x<N; x++){
4          if(A[x] == v)
5              return x;
6      }
7      return -1;
8  }
```

- Como avaliar esse algoritmo?

Pior, Melhor e Caso médio

Considere p algoritmo abaixo:

```
1  #define N 100000
2  int busca(int *A, int v){
3      for(int x = 0; x<N; x++){
4          if(A[x] == v)
5              return x;
6      }
7      return -1;
8  }
```

- Como avaliar esse algoritmo?
- O desempenho depende apenas do tamanho de N?

Pior, Melhor e Caso médio

Considere p algoritmo abaixo:

```
1  #define N 100000
2  int busca(int *A, int v){
3      for(int x = 0; x<N; x++){
4          if(A[x] == v)
5              return x;
6      }
7      return -1;
8  }
```

- Como avaliar esse algoritmo?
- O desempenho depende apenas do tamanho de N?
- O desempenho é modificado pelos valores de entrada?

Pior, Melhor e Caso médio

Pior, Melhor e Caso médio

- **Pior caso** é a função que relaciona o tamanho da entrada *n* com o maior tempo possível para execução deste problema.

Pior, Melhor e Caso médio

- **Pior caso** é a função que relaciona o tamanho da entrada *n* com o maior tempo possível para execução deste problema.
- **Melhor caso** é a função que relaciona o tamanho da entrada *n* com o menor tempo possível para execução deste problema.

Pior, Melhor e Caso médio

- **Pior caso** é a função que relaciona o tamanho da entrada *n* com o maior tempo possível para execução deste problema.
- **Melhor caso** é a função que relaciona o tamanho da entrada *n* com o menor tempo possível para execução deste problema.
- **Caso médio** é a função que relaciona o tamanho da entrada *n* com o tempo médio para execução deste problema. Para isso, é considerado uma *distribuição de probabilidade* das possíveis entradas.

Exemplo:

Qual o pior, melhor e o caso médio para a execução desse algoritmo? Quais suas complexidades?

```
1  #define N 100000
2  int busca(int *A, int v){
3      for(int x = 0; x<N; x++){
4          if(A[x] == v)
5              return x;
6      }
7      return -1;
8  }
```

Exemplo: Complexidade média

Considere:

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 \times \frac{p}{n} + 2 \times \frac{p}{n} + \ldots n \times \frac{p}{n} = \sum_{i=1}^{n} i \times \frac{p}{n} \tag{5}$$

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 \times \frac{p}{n} + 2 \times \frac{p}{n} + \ldots n \times \frac{p}{n} = \sum_{i=1}^{n} i \times \frac{p}{n} \tag{5}$$

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 \times \frac{p}{n} + 2 \times \frac{p}{n} + \ldots n \times \frac{p}{n} = \sum_{i=1}^{n} i \times \frac{p}{n} \tag{5}$$

- O custo do elemento não ser encontrado é:

$$(1 - p)(n + 1) \tag{6}$$

Exemplo: Complexidade média

Somando 6 com 5, temos:

$$
\begin{aligned}
S_{medio} &= (1-p)(n+1) + \sum_{i=1}^{n} i * \frac{p}{n} \\
&= (1-p)(n+1) + \frac{p}{n}\sum_{i=1}^{n} i \\
&= (1-p)(n+1) + \frac{p}{n}\frac{n(n+1)}{2} \\
&= \frac{(n+1)(2-p)}{2}
\end{aligned}
$$

Exemplo: Complexidade média

Somando 6 com 5, temos:

$$
\begin{aligned}
S_{medio} &= (1-p)(n+1) + \sum_{i=1}^{n} i * \frac{p}{n} \\
&= (1-p)(n+1) + \frac{p}{n}\sum_{i=1}^{n} i \\
&= (1-p)(n+1) + \frac{p}{n}\frac{n(n+1)}{2} \\
&= \frac{(n+1)(2-p)}{2}
\end{aligned}
$$

- Considere $p = 1$ (busca bem sucedida), então o custo médio será de $\frac{n+1}{2}$.

Exemplo: Complexidade média

Somando 6 com 5, temos:

$$
\begin{aligned}
S_{medio} &= (1-p)(n+1) + \sum_{i=1}^{n} i * \frac{p}{n} \\
&= (1-p)(n+1) + \frac{p}{n}\sum_{i=1}^{n} i \\
&= (1-p)(n+1) + \frac{p}{n}\frac{n(n+1)}{2} \\
&= \frac{(n+1)(2-p)}{2}
\end{aligned}
$$

- Considere $p = 1$ (busca bem sucedida), então o custo médio será de $\frac{n+1}{2}$.
- Considere $p = 0$ (busca mal sucedida), então o custo médio será de $(n+1)$.

Considerações sobre complexidades

Pontos importantes:

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.
- A complexidade no **pior** caso é tão importante quanto por apresentar o pior cenário possível.

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.
- A complexidade no **pior** caso é tão importante quanto por apresentar o pior cenário possível.
- A complexidade no **melhor** caso não é tão relevante para análise de algoritmos.

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.
- A complexidade no **pior** caso é tão importante quanto por apresentar o pior cenário possível.
- A complexidade no **melhor** caso não é tão relevante para análise de algoritmos.
- A complexidade do caso médio **não** é a média entre o pior caso e o melhor caso.

Complexidades

Pior e Caso médio das estruturas básicas:

Estrutura	n^0 **elem**	**Busca**	**Inserção**	**Remoção**
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Pilha	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Fila	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Tree Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Tree Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Tree Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Bucket Sort	$O(n^2)$	$O(n^2)$	$O(n + k)$

Complexidades

Algoritmo	Pior Caso	Caso Médio	Melhor Caso
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Tree Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Bucket Sort	$O(n^2)$	$O(n^2)$	$O(n + k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$