Análise de Algoritmos

Estruturas de Dados

Prof. David Buzatto

Análise de Algoritmos

- Entender o comportamento de um algoritmo em função da carga de dados que deve ser processada;
- Avaliar o custo em termos de tempo e de espaço;
- Tempo: tempo de processamento;
- Espaço: espaço em memória utilizado.

Análise de Algoritmos

- Dados dois ou mais algoritmos que resolvem o mesmo problema, determinar qual é o melhor ou o pior dependendo da carga de dados;
- A quantidade de dados é relevante, pois um algoritmo julgado pior que outro para grandes quantidades de dados, pode ser melhor para quantidades pequenas.

Análise de Algoritmos

- Complexidade de algoritmos: processo de avaliação dos comandos dos algoritmos com o objetivo de estimar seu tempo de execução em função da carga de dados;
- Carga de dados: tamanho do problema. Exemplos: tamanho de um array, termo de uma série (fibonacci), fatorial de um número, etc.

Melhor caso, pior caso e caso médio

- O tempo de alguns algoritmos dependem não somente da quantidade de dados, mas também do estado desses dados;
- Exemplo: um array desordenado precisa ser ordenado;
- Algoritmo de cálculo do fatorial não depende, mas algoritmos de ordenação dependem.

Melhor caso, pior caso e caso médio

- Três situações:
  - Favorável ➔ melhor caso;
  - Desfavorável ➔ pior caso;
  - Ordinária ➔ caso normal ou caso médio.

Melhor caso, pior caso e caso médio

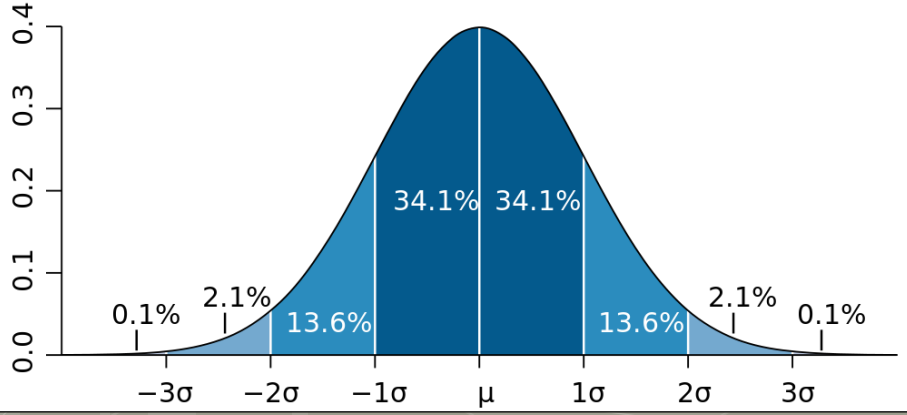
- Exemplo: busca sequencial de um valor em um array que não contém valores repetidos;
  - Se houver alguma tendência do número pesquisado ficar no início do array, o algoritmo executará de forma mais rápida. Melhor caso.

Melhor caso, pior caso e caso médio

- Exemplo: busca sequencial de um valor em um array que não contém valores repetidos;
  - Se o valor pesquisado não existir no array ou houver tendência do valor ficar no fim do array, o algoritmo executará de forma mais lenta. Pior caso.

Melhor caso, pior caso e caso médio

- Exemplo: busca sequencial de um valor em um array que não contém valores repetidos;
  - Se não houver tendência, o algoritmo percorrerá em média a metade dos itens do array. Caso médio (distribuição normal).



Complexidade Assintótica

- Análise da complexidade de algoritmos onde o tamanho do problema tende ao infinito (valores muito grandes).
- O que importa são cargas muito grandes de dados, não de 100, 200 ou 300 itens, mas sim de milhares ou milhões de elementos.
- Exemplos: consultas em bases de dados, ordenação de uma lista telefônica, etc.

Aspectos Formais e Notação

- **n**: tamanho do problema. Valor que o tempo de execução depende. É sempre maior ou igual a zero. Exemplos: tamanho de um array, fatorial de um número, quantidade de nós de uma árvore binária, etc.

Aspectos Formais e Notação

- **T(n)**: função que reflete o tempo de execução de um algoritmo de carga **n**. É sempre positiva. A avaliação das características de T(n) é a análise de complexidade do algorimo.

Aspectos Formais e Notação

- **O(f(n))**: notação de complexidade. Representa as características globais sobre o comportamento de um algoritmo.
- Exemplos:
  - **O(n)**;
  - **O(n log n)**; (considerar logaritmo de base 2)
  - **O(n^2)**, etc.

Aspectos Formais e Notação

- Exemplo:

$$\text{Para } O(n^2),\ T(n) = n^2$$

Sendo assim:

se n = 10, $T(10) = 10^2 = 100$

se n = 100, $T(100) = 100^2 = 10000$

Como:

$$T(n) = n2$$

$$T(10n) = (10n)^2 = 100n^2 = 100T(n)$$

Aspectos Formais e Notação

$$T(n) = n2$$
$$T(10n) = (10n)^2 = 100n^2 = 100T(n)$$

Conclusão: para um algoritmo de complexidade $O(n^2)$, o aumento da carga em 10 vezes, implica no aumento no tempo em 100 vezes!

Aspectos Formais e Notação

- T(n) pertence a O(f(n)) se e somente se $T(n) \leq cf(n)$, para todos os valores de $n \geq n_0$, com c e n_0 positivos e arbitrários.

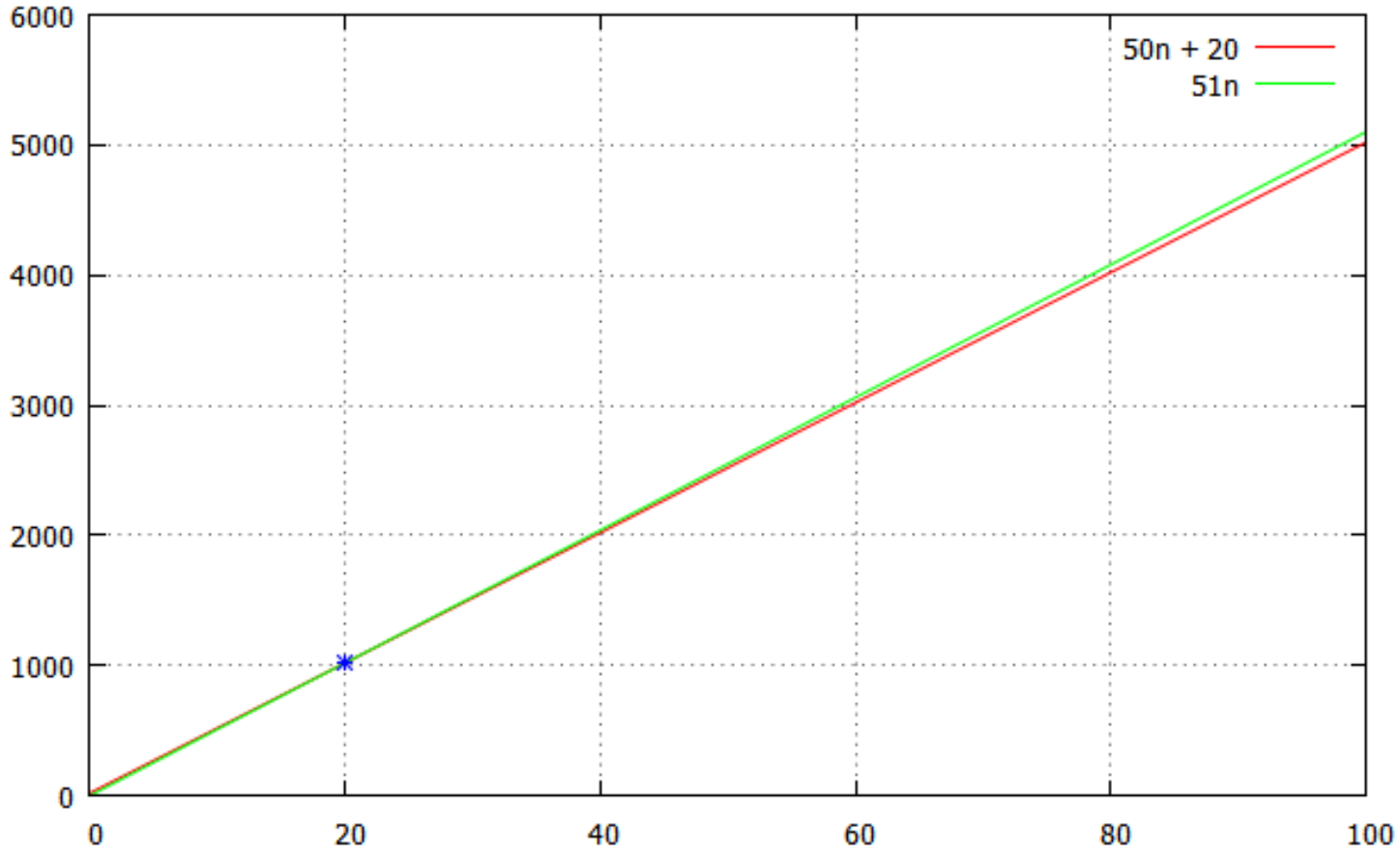
Aspectos Formais e Notação

- "Se, a partir de um dado tamanho de problema (n), o tempo de execução do algoritmo (T(n)) for limitado superiormente (ou seja, for menor ou igual) por uma função cf(n), então a notação de qualificação O(f(n)) pode ser usada para indicar a complexidade do algoritmo"

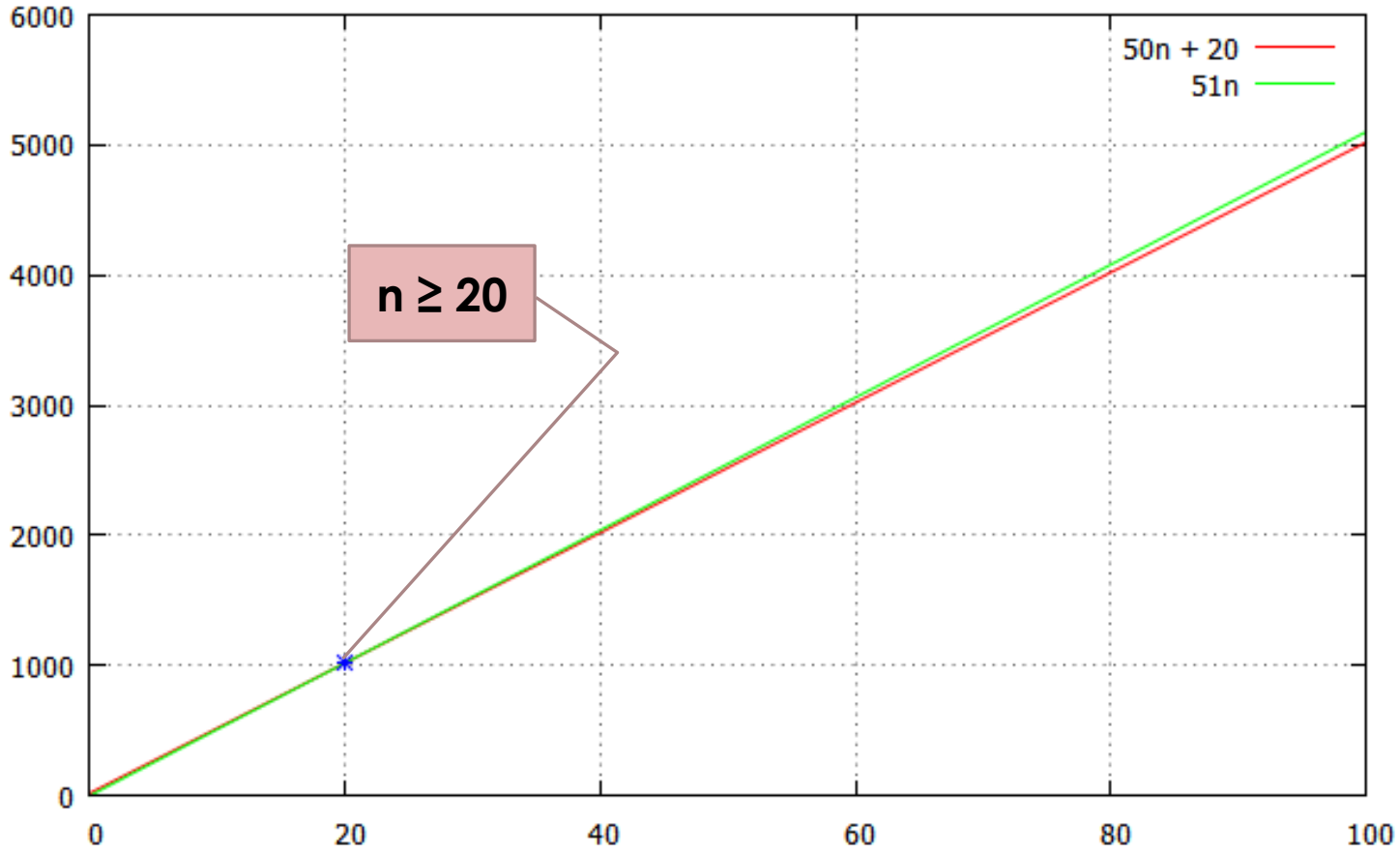
Aspectos Formais e Notação

- Exemplo 1: Se $T(n) = 50n + 20$, mostre que $T(n)$ pertence a $O(n)$.
- Para $T(n)$ pertencer a $O(n)$, $50n + 20 \leq cn$
- Deve-se mostrar que existem valores para c e n_0 que façam com que $T(n) \leq cn$ seja válido para $n \geq n_0$

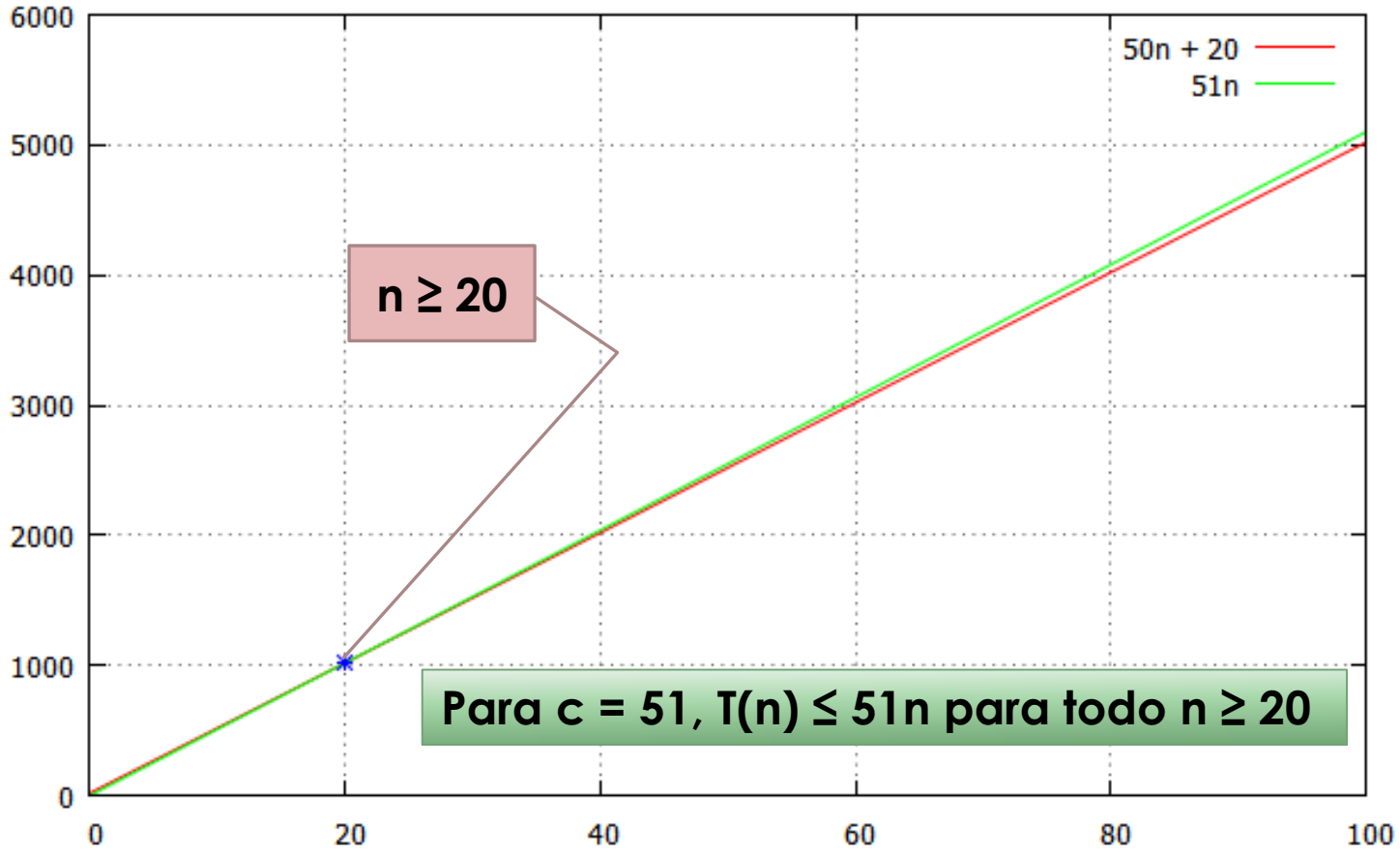
Aspectos Formais e Notação



Aspectos Formais e Notação



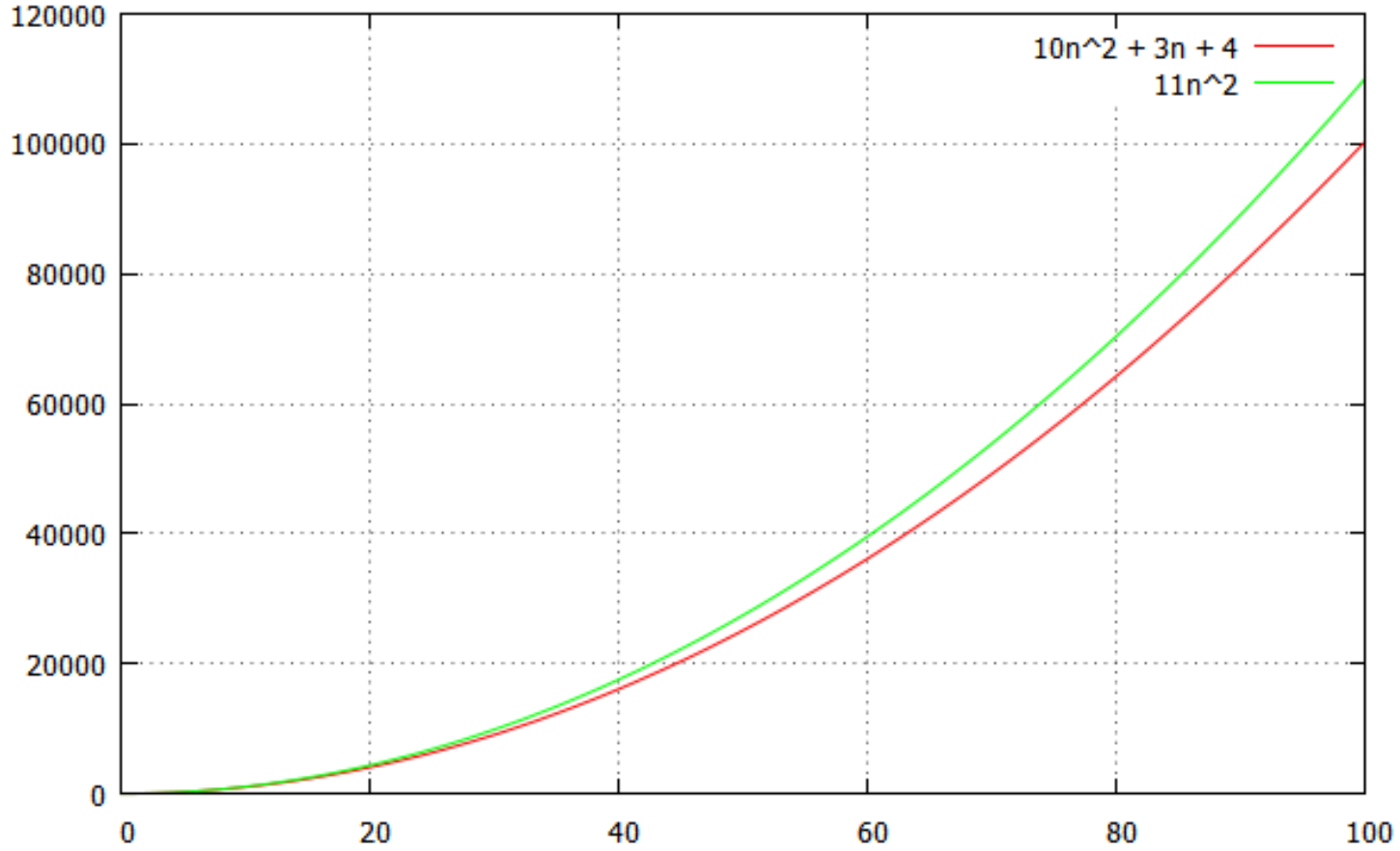
Aspectos Formais e Notação



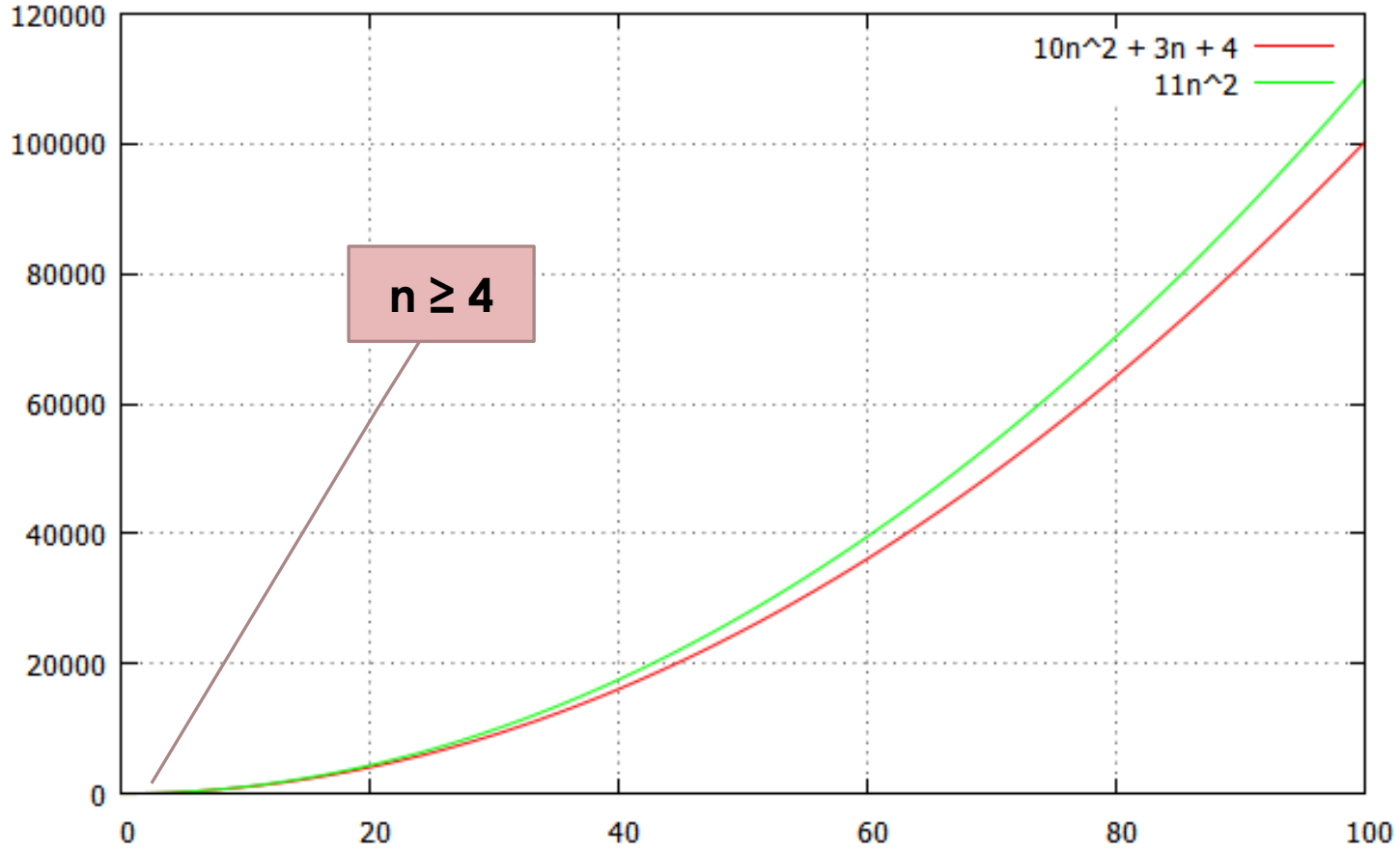
Aspectos Formais e Notação

- Exemplo 2: Se $T(n) = 10n^2 + 3n + 4$, mostre que $T(n)$ pertence a $O(n^2)$.
- Para $T(n)$ pertencer a $O(n^2)$, $10n^2 + 3n + 4 \leq cn^2$
- Deve-se mostrar que existem valores para c e n_0 que façam com que $T(n) \leq cn^2$ seja válido para $n \geq n_0$

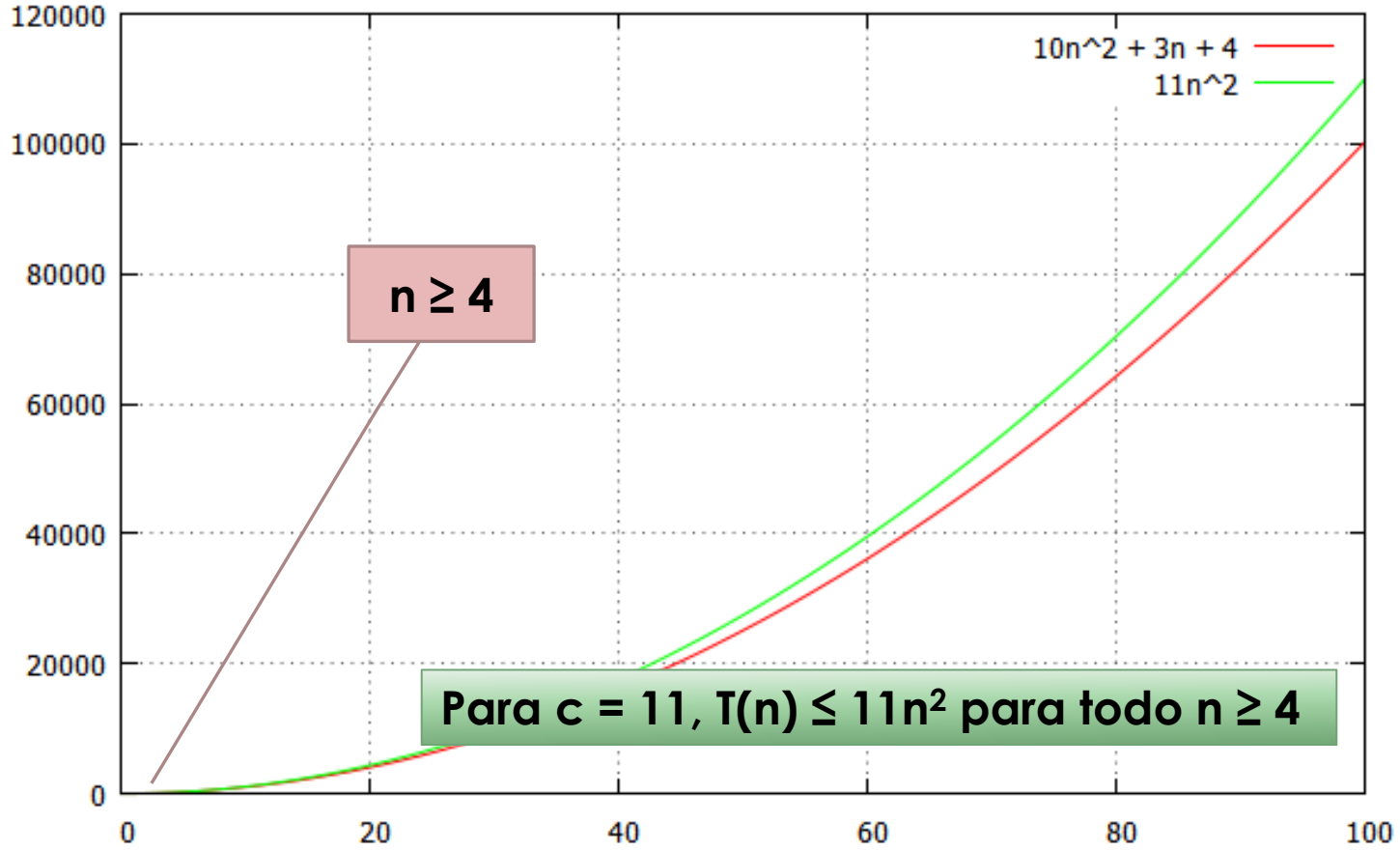
Aspectos Formais e Notação



Aspectos Formais e Notação



Aspectos Formais e Notação

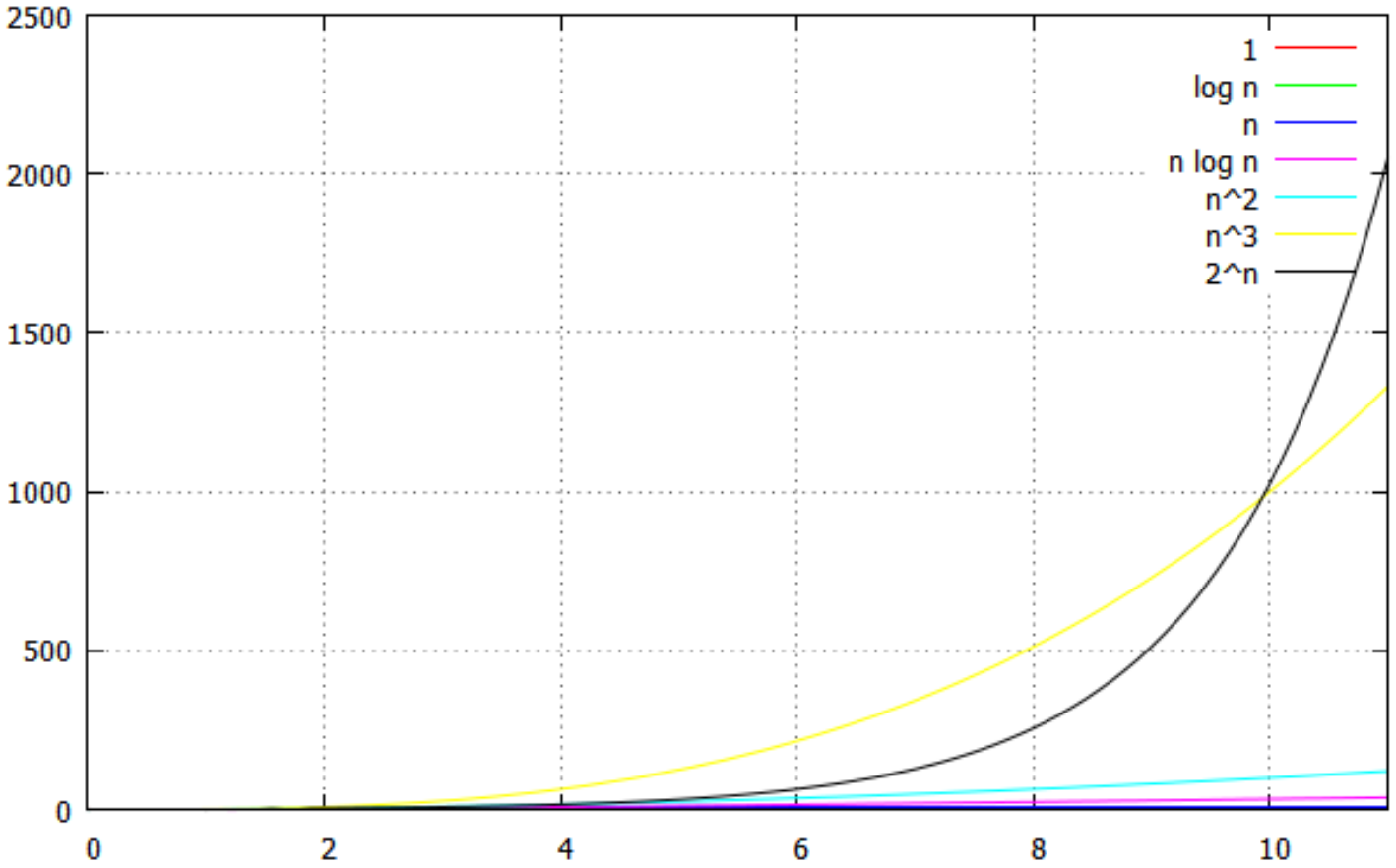


Aspectos Formais e Notação

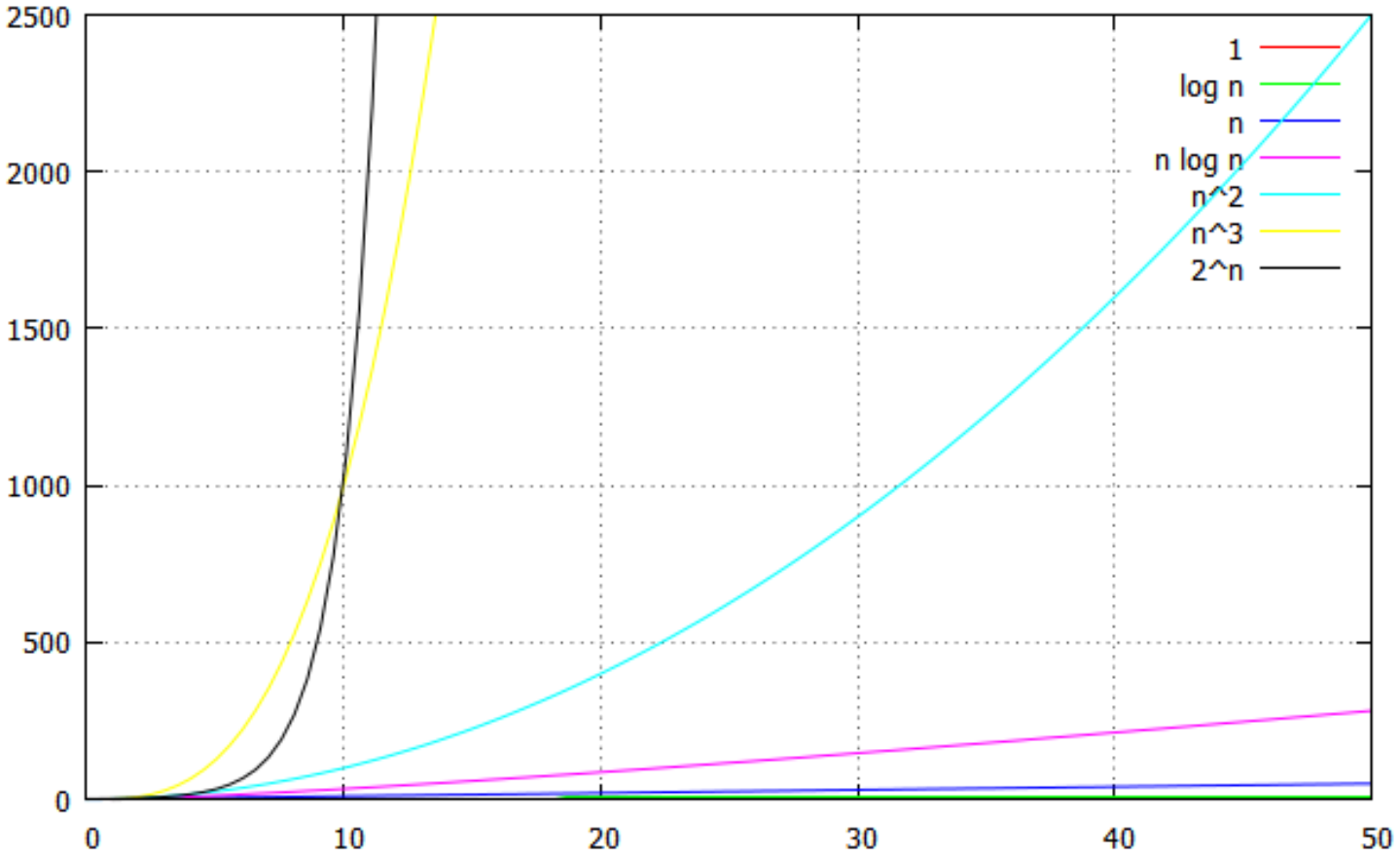
Ordem relativa	Função
1	1
2	log n
3	n
4	n log n
5	n^2
6	n^3
7	2^n

Funções mais usadas para indicar a complexidade de algoritmos.
Obs: a notação log indica logaritmo de base 2

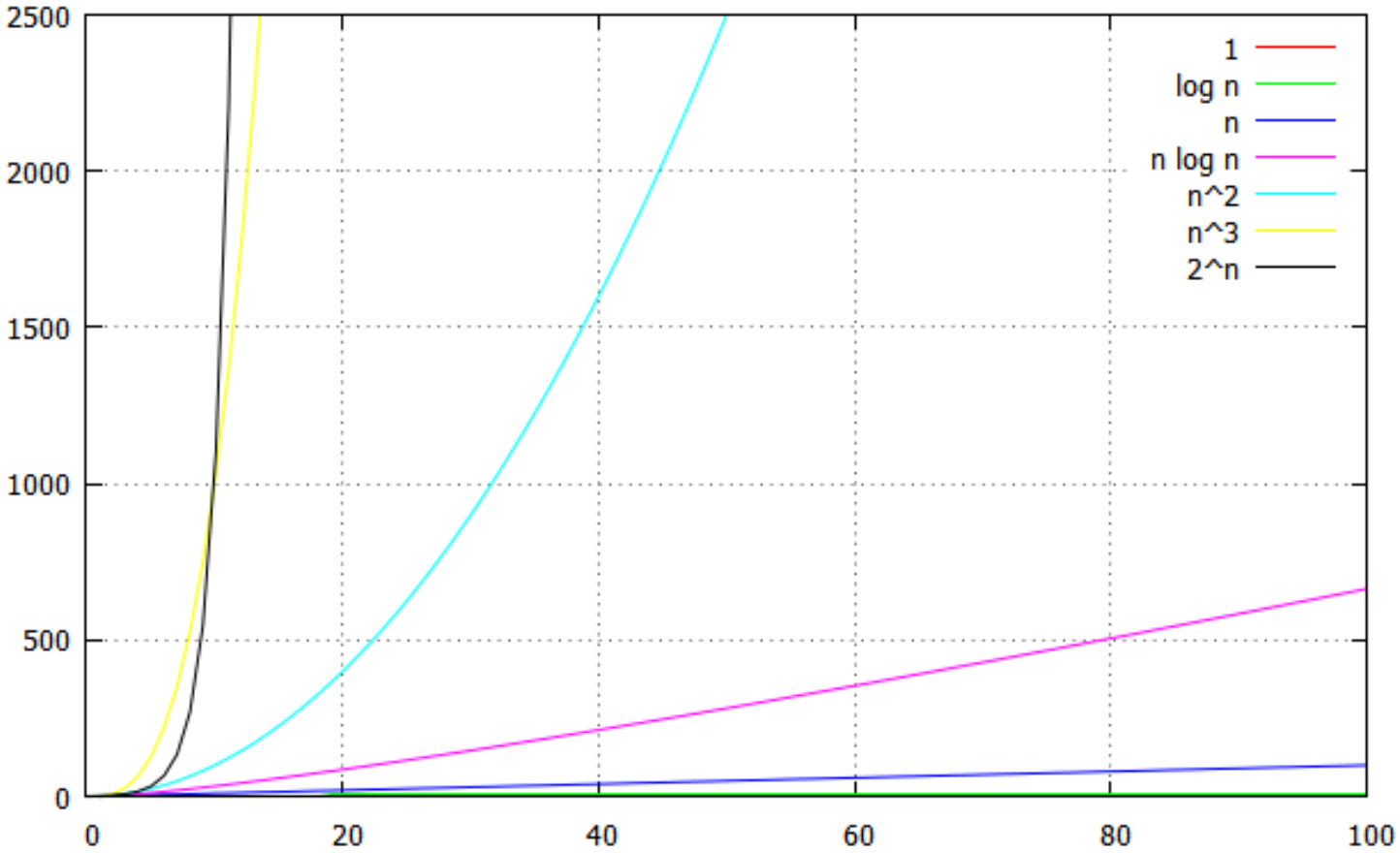
Aspectos Formais e Notação



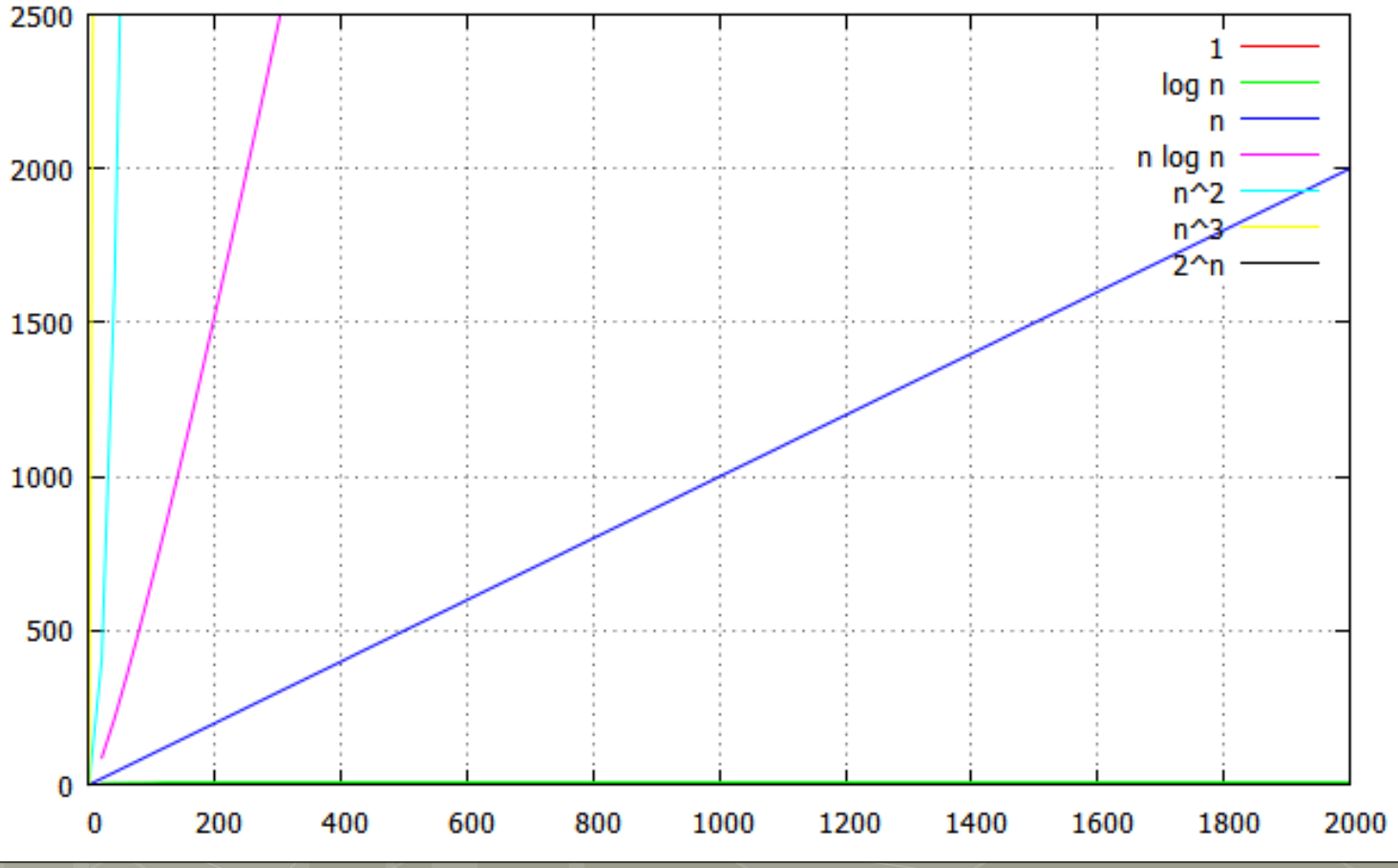
Aspectos Formais e Notação



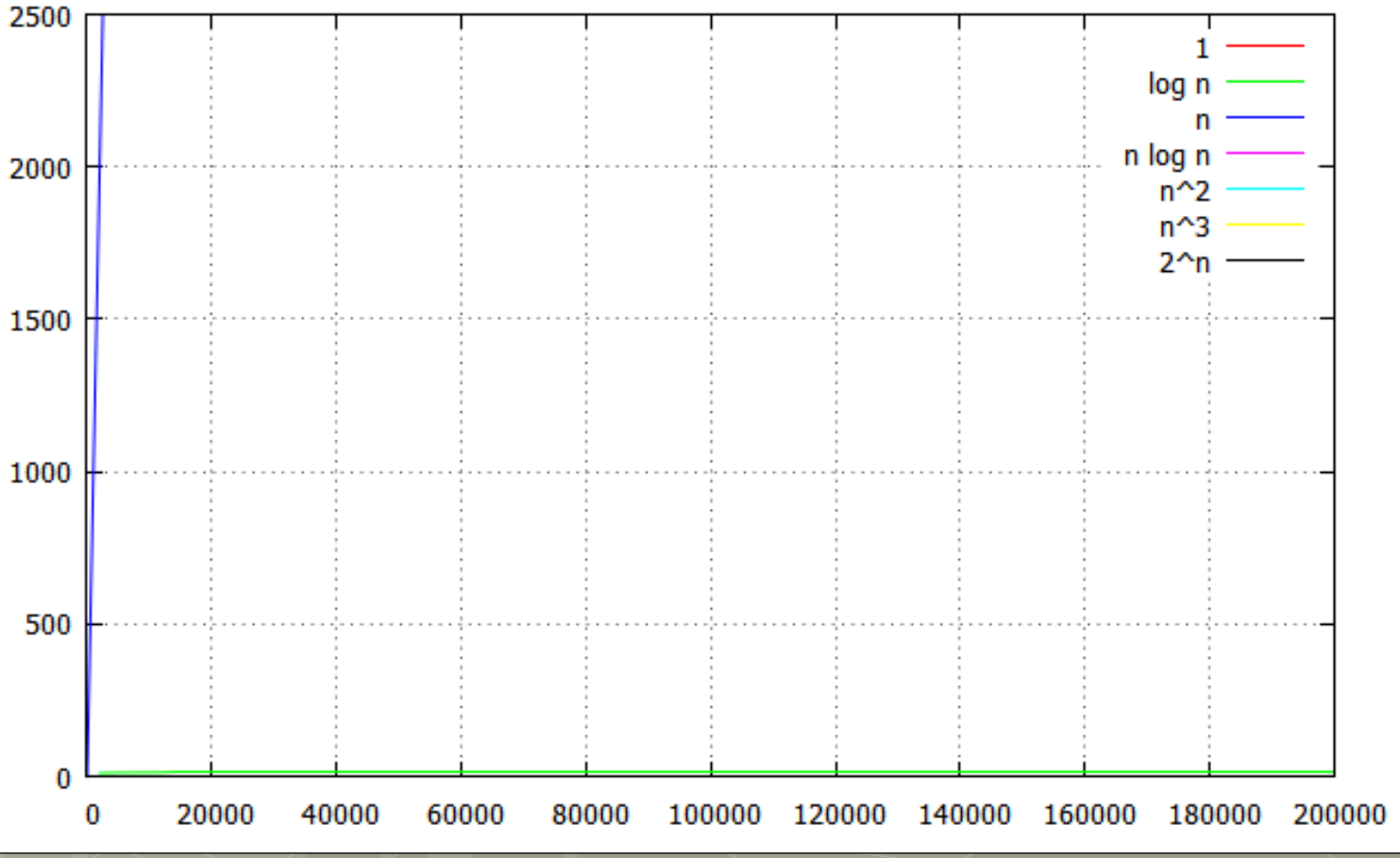
Aspectos Formais e Notação



Aspectos Formais e Notação



Aspectos Formais e Notação



O Impacto da Complexidade

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
1	1,00 µs	< 1,00 µs	1,00 µs	< 1,00 µs	1,00 µs	2,00 µs
10	1,00 µs	3,32 µs	10,00 µs	33,22 µs	100,00 µs	1,02 ms
20	1,00 µs	4,32 µs	20,00 µs	86,44 µs	400,00 µs	1,05 s
30	1,00 µs	4,91 µs	30,00 µs	147,21 µs	900,00 µs	17,90 min
50	1,00 µs	5,64 µs	50,00 µs	282,19 µs	2,50 ms	13031,25 dias
100	1,00 µs	6,64 µs	100,00 µs	664,39 µs	10,00 ms	$4{,}02\times10^{13}$ séculos
500	1,00 µs	8,97 µs	500,00 µs	4,48 ms	250,00 ms	$1{,}04\times10^{131}$ milênios
1.000	1,00 µs	9,97 µs	1,00 ms	9,97 ms	1,00 s	$3{,}40\times10^{281}$ milênios
5.000	1,00 µs	12,29 µs	5,00 ms	61,44 ms	25,00 s	~∞
10.000	1,00 µs	13,29 µs	10,00 ms	132,88 ms	1,67 min	~∞
100.000	1,00 µs	16,61 µs	100,00 ms	1,66 s	166,67 min	~∞

Tempos de execução para algoritmos de complexidades variadas, supondo um computador que execute uma instrução por microssegundo

O Impacto da Complexidade

T(n)	Tamanho de problema resolvido	100 vezes mais rápido	1000 vezes mais rápido	1.000.000.000 vezes mais rápido
n	K	100 K	1000 K	10^9 K
n^2	K	10 K	31,6 K	31623 K
n^3	K	4,6 K	10 K	1000 K
n^5	K	2,5 K	3,9 K	63 K
2^n	K	K + 6,6	K + 9,9	K + 30

Tamanho do problema que seria executado em um período fixo de tempo, caso a velocidade de processamento fosse aumentada

O Impacto da Cor

Se o processador original for trocado por um 100 vezes mais rápido, eu consigo resolver um problema 100 vezes maior que o original no mesmo tempo.

T(n)	Tamanho de problema resolvido	100 vezes mais rápido	[illegible] vezes mais rápido	1.000.000.000 vezes mais rápido
n	K	100 K	1000 K	10^9 K
n^2	K	10 K	31,6 K	31623 K
n^3	K	4,6 K	10 K	1000 K
n^5	K	2,5 K	3,9 K	63 K
2^n	K	K + 6,6	K + 9,9	K + 30

Tamanho do problema que seria executado em um período fixo de tempo, caso a velocidade de processamento fosse aumentada

O Modelo de Computador e os Comandos

- Algoritmo:

$$c \leftarrow a + b$$

- Considerar:
  - Tempo de acesso à memória (a e b);
  - Realização da operação (soma);
  - Armazenamento do resultado em local temporário;
  - Atribuição do valor obtido em c.

O Modelo de Computador e os Comandos

- O tempo da realização dessas operações pode variar de máquina para máquina;
- Detalhes muito finos podem ser desconsiderados na análise de algoritmos;

O Modelo de Computador e os Comandos

- Sendo assim, por mais diferentes que as operações sejam, devemos considerar que cada uma gastará apenas uma unidade de tempo;
- No algoritmo "c ← a + b", são gastas 4 unidades de tempo (ut):
  - Duas para a obtenção de a e b;
  - Uma para a soma;
  - Uma para a atribuição.

O Modelo de Computador e os Comandos

- Mesmo o nível de detalhe apresentado anteriormente será desnecessário, visto que estamos preocupados com a "cara" da função de tempo e não com os valores dos coeficientes que a compõe.

Trechos de Tempo Constante

- "Dado o valor de um número inteiro positivo, calcular e apresentar a soma de todos os inteiros positivos menores ou iguais a ele."

Trechos de Tempo Constante

```
{soma de positivos até um dado valor}

algoritmo
    declare valor, soma: inteiro

    leia(valor);
    soma ← valor * ( ( valor + 1 ) / 2 )
    escreva(soma)
fim-algoritmo
```

Trechos de Tempo Constante

- Qual o tamanho do problema?
  - Verificando o código, é possível verificar que independente do valor digitado pelo usuário, o algoritmo consome o mesmo tempo.

Trechos de Tempo Constante

```
{soma de positivos até um dado valor}

algoritmo
    declare valor, soma: inteiro

    leia(valor);
    soma ← valor * ( ( valor + 1 ) / 2 )
    escreva(soma)
fim-algoritmo
```

Trechos de Tempo Constante

```
{soma de positivos até um dado valor}

algoritmo
    declare valor, soma: inteiro

    leia(valor);
    soma ← valor * ( ( valor + 1 ) / 2 )
    escreva(soma)
fim-algoritmo
```

2ut

Trechos de Tempo Constante

```
{soma de positivos até um dado valor}

algoritmo
    declare valor, soma: inteiro
    leia(valor);
    soma ← valor * ( ( valor + 1 ) / 2 )
    escreva(soma)
fim-algoritmo
```

2ut

6ut

Trechos de Tempo Constante

```
{soma de positivos até um dado valor}

algoritmo
    declare valor, soma: inteiro
    leia(valor);
    soma ← valor * ( ( valor + 1 ) / 2 )
    escreva(soma)
fim-algoritmo
```

2ut

6ut

2ut

Trechos de Tempo Constante

- Sendo assim:
  - 2ut + 6ut + 2ut = 10ut;
  - T(n) = 10ut;
- Como não há tamanho do problema, o consumo de tempo do algoritmo é constante, pois não há influência do tamanho do problema.
- T(n) = k, sendo k uma constante
- T(n) é O(1)

Trechos com Repetições

- Tipo principal de estrutura que influencia no consumo de tempo;
- Novamente, considerando o problema de obter a soma de todos os inteiros menores ou iguais a um inteiro desejado.

Trechos com Repetições

```
{soma de positivos até um dado valor}

algoritmo
    declare i, valor, soma: inteiro

    leia(valor)
    soma ← 0

    para i ← 1 até valor faça
        soma ← soma + i
    fim-para

    escreva(soma)

fim-algoritmo
```

Trechos com Repetições

```
{soma de positivos até um dado valor}

algoritmo
    declare i, soma: inteiro

    leia(valor)
    soma ← 0

    para i ← 1 até valor faça
        soma ← soma + i
    fim-para

    escreva(soma)

fim-algoritmo
```

2ut

1ut

2ut

Trechos com Repetições

```
{soma de positivos até um dado valor}

algoritmo
    declare i, valor, soma: inteiro

    leia(valor)
    soma ← 0

    para i ← 1 até valor faça
        soma ← soma + i
    fim-para

    escreva(soma)

fim-algoritmo
```

Trechos de instruções

```
{soma de positivo

algoritmo
    declare i, va

    leia(valor)
    soma ← 0

    para i ← 1 até valor faça
        soma ← soma + i
    fim-para

    escreva(soma)

fim-algoritmo
```

1ut

i incrementa de 1 até o tamanho do problema, sendo assim, n incrementos, gastando 3ut cada um:
3n ut

Trechos c ções

```
{soma de positivo

algoritmo
    declare i, va

    leia(valor)
    soma ← 0

    para i ← 1 até valor faça
        soma ← soma + i
    fim-para

    escreva(soma)

fim-algoritmo
```

1ut

i incrementa de 1 até o tamanho do problema, sendo assim, n incrementos, gastando 3ut cada um:
3n ut

além do incremento, i é comparado com n, verificando se o laço deve parar. são feitas n+1 comparações, gastando 3ut cada uma.
3n + 3ut

Trechos c ções

```
{soma de positivo

algoritmo
    declare i, va

    valor)
    soma ← 0

    para i ← 1 até valor faça
        soma ← soma + i
    fim-para

    escreva(soma)

fim-algoritmo
```

1ut

i incrementa de 1 até o tamanho do problema, sendo assim, n incre gastando 3u um: 3n ut

Essa linha gasta 4ut. Ela é executada n vezes, sendo assim: 4n ut

comparado com n, verificando se o laço deve parar. são feitas n+1 comparações, gastando 3ut cada uma: 3n + 3ut

Trechos com Repetições

- Somando-se os tempos em função de n:
  - $5 + 1 + 3n + 3n + 3 + 4n = 10n + 9$
- Sendo assim, $T(n) = 10n + 9$
- $T(n)$ é $O(n)$
- Qual algoritmo é melhor?

Trechos com Repetições

```
leia(n)
para i ← 1 até n/2 faça
    escreva(i)
fim-para
```

- Considerando n par:

  2ut, 1ut, (n/2*3ut), ((n/2 + 1)*3ut), (n/2*2ut)

$$T(n) = 2 + 1 + \frac{3n}{2} + \frac{3n}{2} + 3 + n$$

Trechos com Repetições

$$\boldsymbol{T(n) = 2 + 1 + \frac{3n}{2} + \frac{3n}{2} + 3 + n}$$

$$T(n) = 3 + \frac{6n}{2} + 3 + n$$

$$T(n) = 3n + n + 6$$

$$T(n) = 4n + 6$$

Trechos com Repetições

$$T(n) = 4n + 6$$

- T(n) é O(n)
- Esse algoritmo tem **comportamento** igual ao algoritmo anterior (10n + 9) e não o tempo de execução, pois este é praticamente duas vezes mais rápido.

Trechos com Repetições

```
leia(n)
para i ← 1 até n faça
     para j ← 1 até n faça
          escreva(i*j)
     fim-para
fim-para
```

2, 1, (n*3), ((n+1)*3), n*(1+(n*3)+((n+1)*3)+(n*4))

Trechos com Repetições

2, 1, (n*3), ((n+1)*3), n*(1+(n*3)+((n+1)*3)+(n*4))

$$\boldsymbol{T(n) = 2 + 1 + 3n + 3n + 3 + n(1 + 3n + 3n + 3 + 4n)}$$

$$T(n) = 6n + 6 + n(10n + 4)$$

$$T(n) = 6n + 6 + 10n^2 + 4n$$

$$T(n) = 10n^2 + 10n + 6$$

Trechos com Repetições

$$T(n) = 10n^2 + 10n + 6$$

- T(n) é $O(n^2)$

Trechos com Repetições

- O aninhamento de repetições que tenham as características mostradas acarretam no aumento do polinômio, ou seja:

Outras Considerações

- A análise de algoritmos apresentada leva em consideração o comportamento dos algoritmos para cargas de dados grandes, entretanto isso não significa que um algoritmo pertencente a $O(n^2)$ seja pior que um algoritmo $O(n)$, visto que a avaliação feita é geral.

Outras Considerações