ALGORITMOS E ESTRUTURAS DE DADOS I

Prof. Caio César de Freitas Dantas

Complexidade de Algoritmo

- Algoritmo é um processo sistemático para a resolução de um problema.

- São sequências de passos bem definidos, detalhados e finitos, que quando executados realizam uma tarefa específica.

- Estudo de algoritmos envolve 2 aspectos básicos: correção e análise.
    - Correção: exatidão do método empregado (prova matemática).
    - Análise: avaliar a eficiência do algoritmo em termos dos recursos (memória e tempo de execução) utilizados.

Complexidade de Algoritmo

- Algoritmo: Entrada (dados) => Processamento => Saída

- Por ser finito, um algoritmo deve:
  - Iniciar;
  - Processar por um determinado tempo;
  - Terminar;

Em geral, tendemos a comparar dois algoritmos distintos, que realizam uma mesma tarefa, pelo tempo necessário à sua execução.

Complexidade de Algoritmo

- Diferentes computadores, muitas vezes com hardwares idênticos, podem levar tempos diferentes para processar um mesmo algoritmo;
    - Quando trocamos algum, ou mais, itens de hardware, esta diferença tende a aumentar ainda mais;
- Por tanto, avaliar o desempenho de um algoritmo com base apenas no tempo de execução deste, é ineficiente.

Complexidade de Algoritmo

- A Complexidade de Algoritmos estuda e define quanto eficiente é um algoritmo em relação ao número de operações (passos do algoritmo) necessárias para finalizar a tarefa.

- Os princípios básicos de Complexidade é uma ferramenta útil para escolha e/ou desenvolvimento do melhor algoritmo a ser utilizado para resolver determinado problema.

Complexidade de Algoritmo

Critérios pra medir a qualidade de um software:

- Lado do usuário ou cliente:
    - Interface
    - Robustez
    - Compatibilidade
    - **Desempenho (rapidez)**
    - **Consumo de recursos (ex. Memória)**

- Lado do desenvolvedor ou fornecedor:
    - Portabilidade
    - Clareza
    - Reuso

Complexidade de Algoritmo

Uma das formas mais simples de avaliar um algoritmo é através da análise empírica: rodar 2 ou mais algoritmos e verificar qual o mais rápido.

Desafios da análise empírica:

- Desenvolver uma implementação correta e completa.
- Determinar a natureza dos dados de entrada e de outros fatores que têm influência no experimento.

Apesar do hardware influenciar no tempo de execução de um algoritmo, existe ainda um segundo parâmetro que pode também influenciar este tempo:

– O Conjunto de Dados!

Complexidade de Algoritmo

Tipicamente temos 3 escolhas básicas de dados:

- Reais: similar as entradas normais para o algoritmo; realmente mede o custo do programa em uso.
- Randômicos: gerados aleatoriamente sem se preocupar se são dados reais; testa o algoritmo em si.
- Problemáticos: dados manipulados para simular situações anômalas; garante que o programa sabe lidar com qualquer entrada.

Complexidade de Algoritmo

Em alguns casos a análise matemática é necessária:

- Se a análise experimental começar a consumir uma quantidade significando de tempo então é o caso de realizar análise matemática.

- É necessário alguma indicação de eficiência antes de qualquer investimento de desenvolvimento.

Análise Matemática é uma forma de avaliar um algoritmo que pode ser mais informativa e menos cara de se executar.

Complexidade de Algoritmo

Em geral, algoritmos servem para processar conjuntos de dados com tamanho indeterminado:

– Listas;
– Filas;
– Pilhas;
– Tabelas;
– Imagens;
– Entre outros.

Complexidade de Algoritmo

Exemplo:

Para somar todos os elementos de uma lista:

- Quando a lista tiver apenas 1 elemento:
  – Uma operação de soma!

- Quanto a lista tiver 1.000.000 de elementos:
  – 1.000.000 de somas!

Complexidade de Algoritmo

Por padrão, para denotar o tamanho do conjunto de dados a ser processado, chamamos este número de dados de:

n

Criamos, então, uma “função de complexidade do algoritmo”, que irá relacionar o número de instruções utilizadas por um algoritmo – **T(n)** – com o tamanho do conjunto de dados – **n**:

$$T(n) = n$$

Neste exemplo, para cada **n** dados, teremos **n** operações.

Complexidade de Algoritmo

As Complexidades mais comuns são:

$\log_2 n$

n

$n \log_2 n$

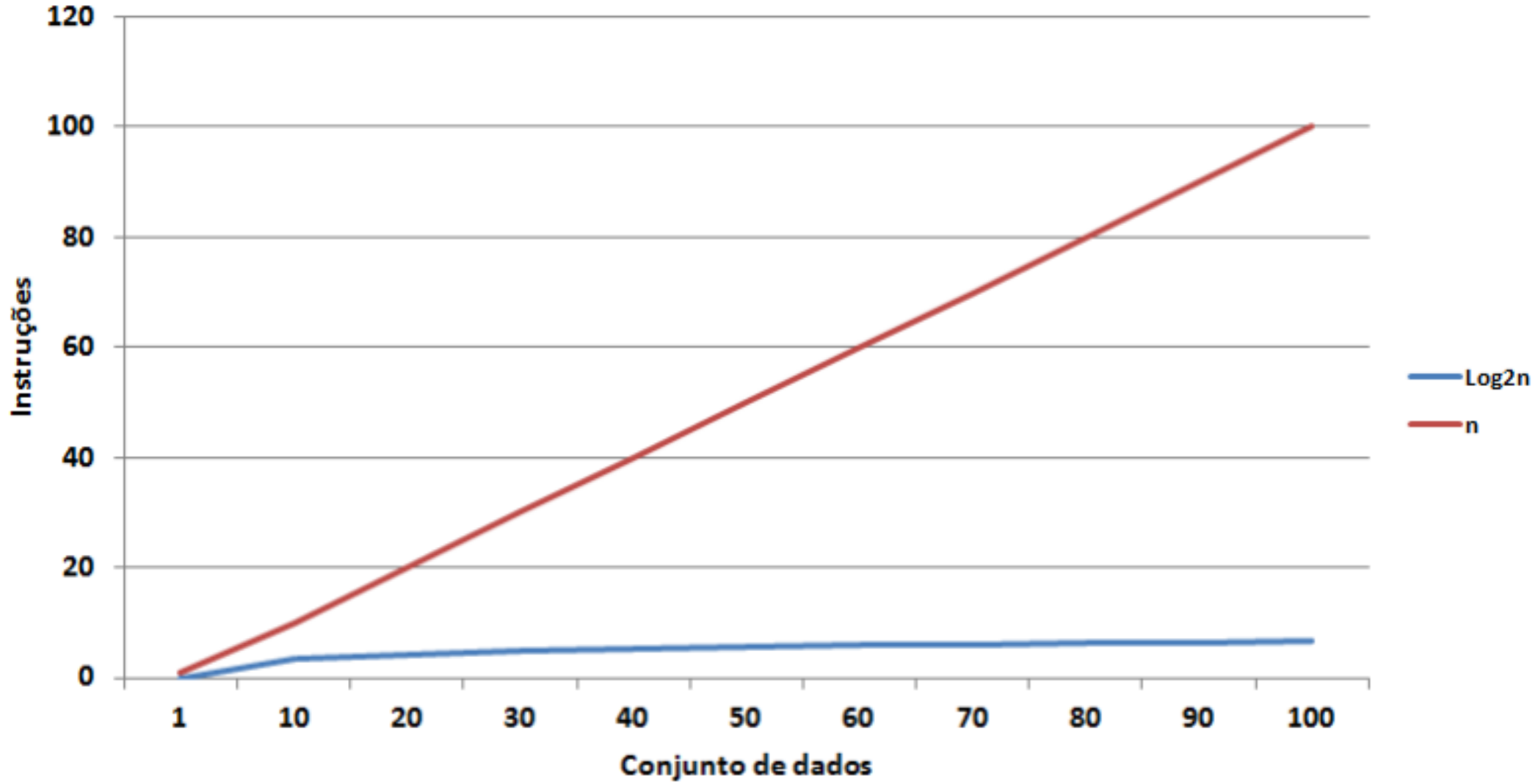
n^2

n^3

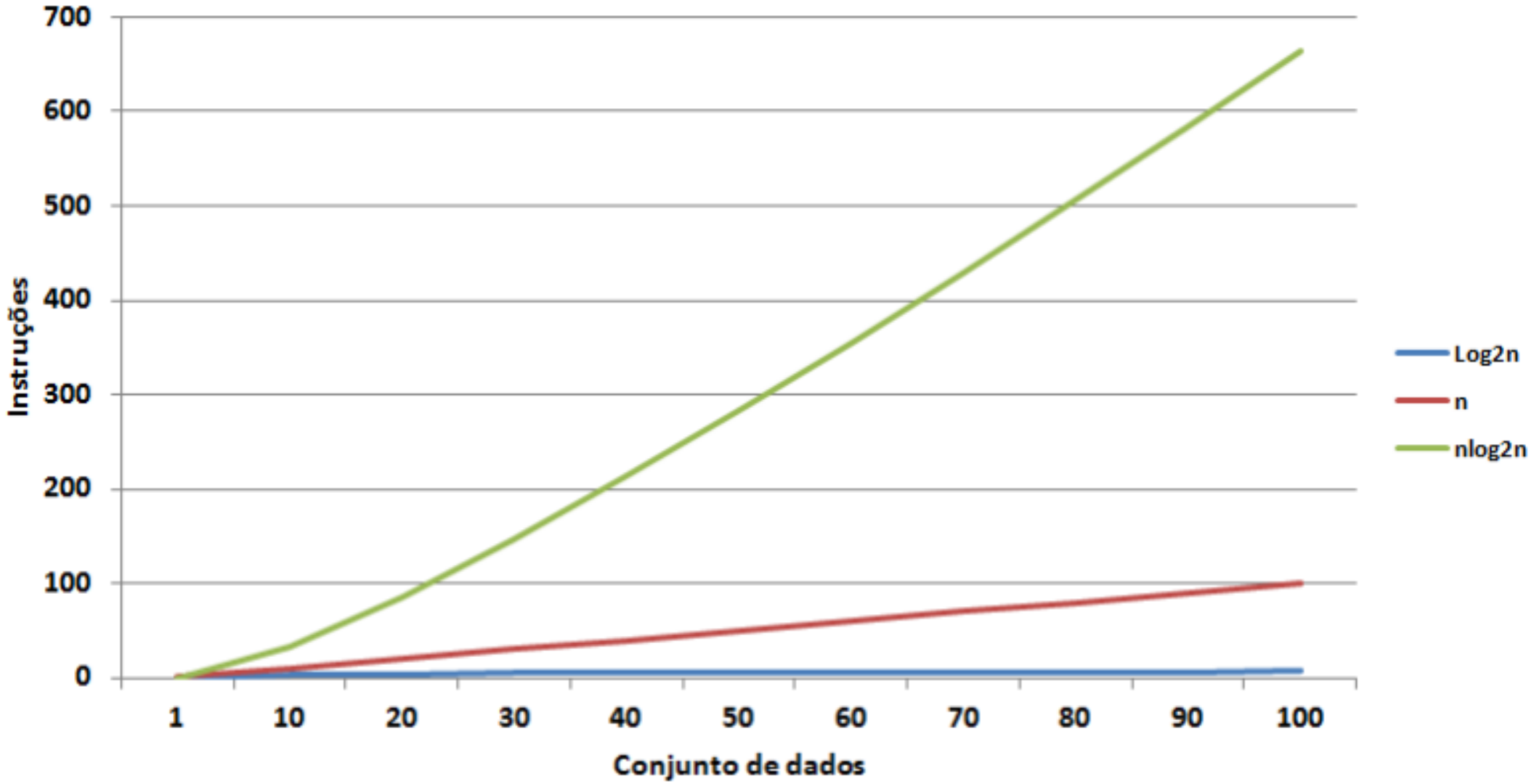
2^n

3^n

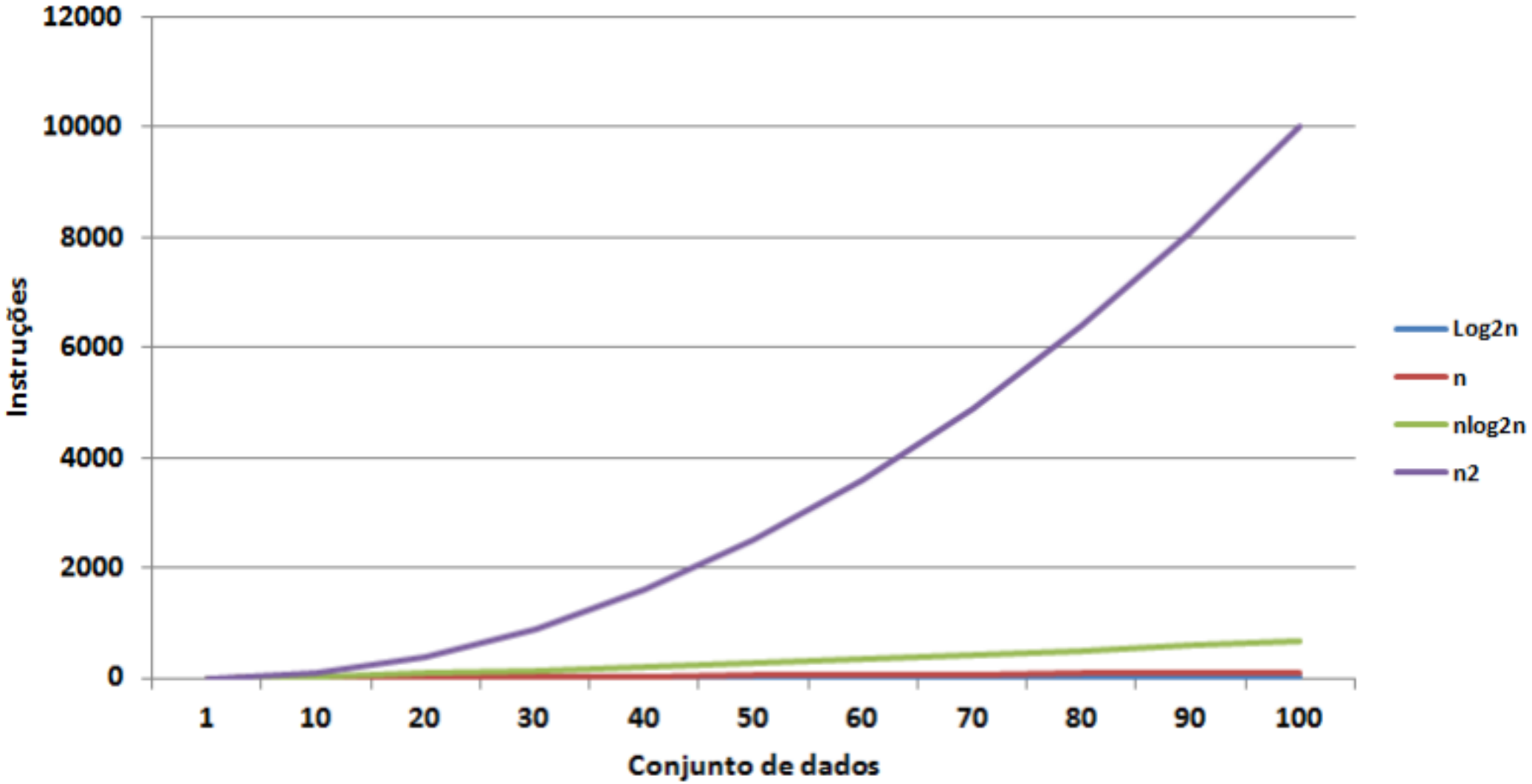
Complexidade de Algoritmo



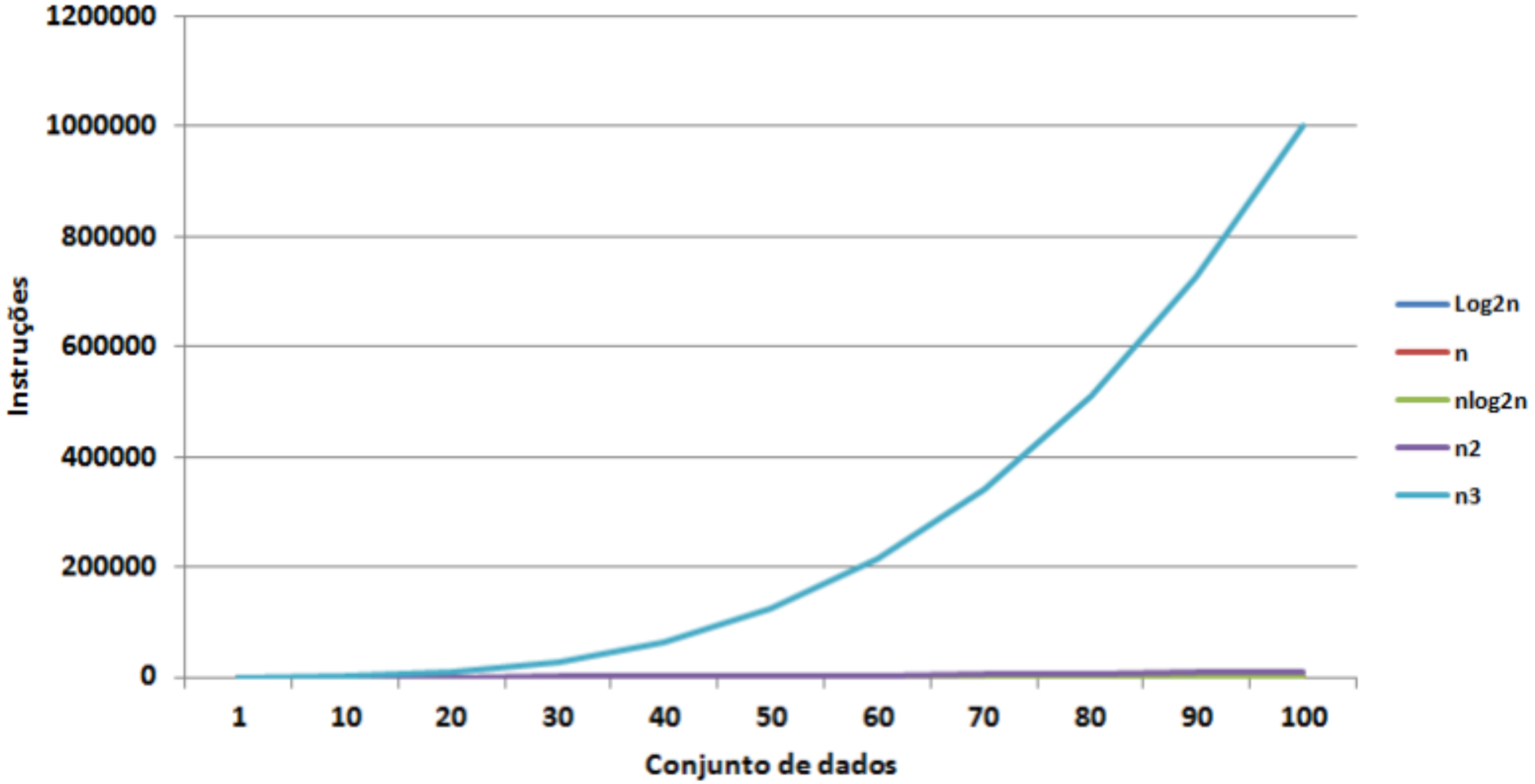
Complexidade de Algoritmo



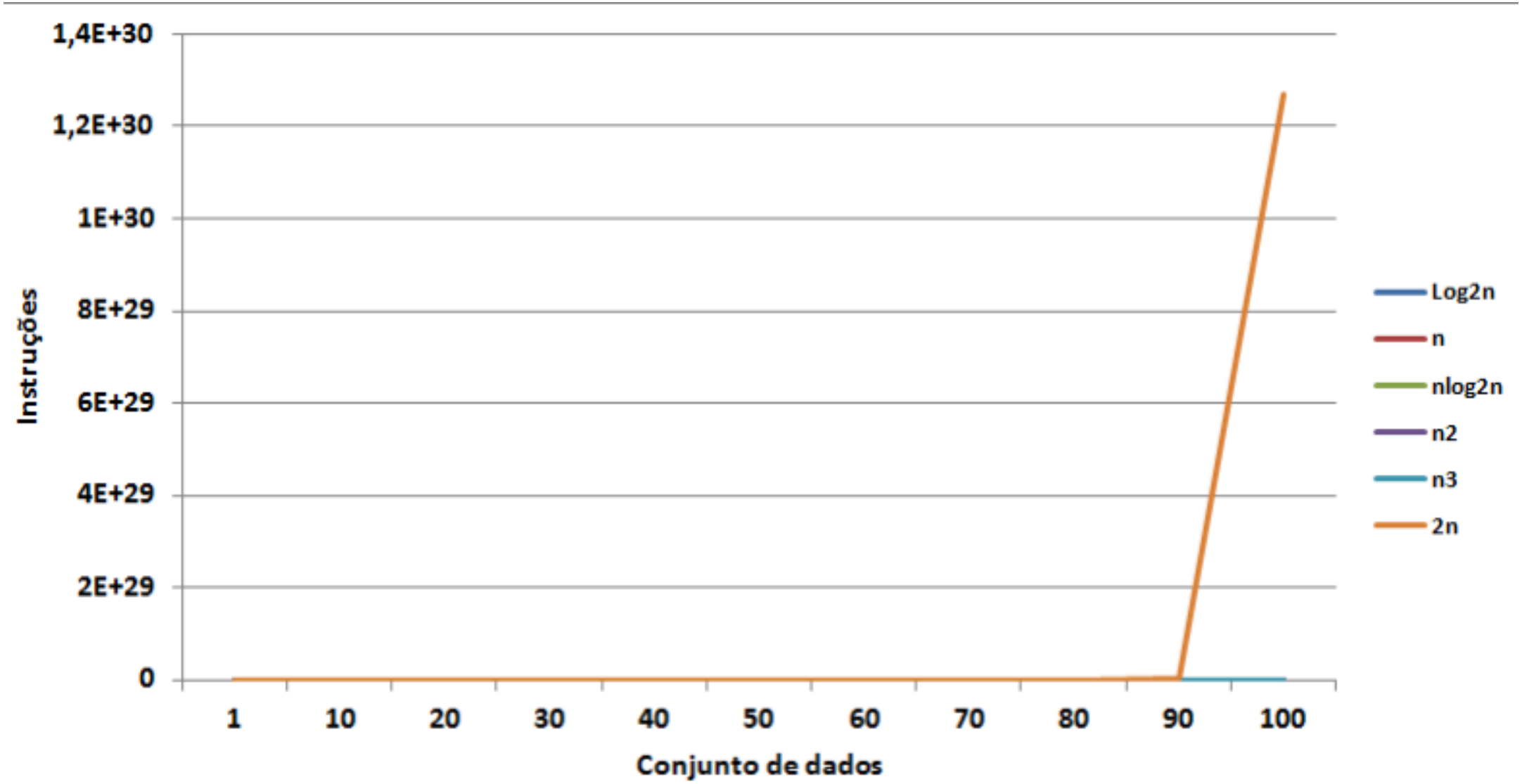
Complexidade de Algoritmo



Complexidade de Algoritmo



Complexidade de Algoritmo



Complexidade de Algoritmo

– Qual é o T(n) para a pesquisa de um elemento em uma lista?

– Qual é o T(n) para inserir um elemento em uma lista?

Complexidade de Algoritmo

– Qual é o T(n) para a pesquisa de um elemento em uma lista?

- O primeiro elemento ser o procurado (melhor caso);
- O elemento central ser o procurado (caso médio);
- O último elemento ser o procurado (pior caso);

– Qual é o T(n) para inserir um elemento em uma lista?

- Uma instrução para criar o próximo elemento;

Complexidade de Algoritmo

Cenários possíveis:

- Melhor Caso:

Descreve o menor número de instruções possíveis de ocorrer para um determinado algoritmo (conforme os dados utilizados);

No exemplo de pesquisa em uma lista, o melhor cenário existe quando o primeiro elemento é o procurado:

Neste caso, T(n) = 1, utiliza-se a notação:

$$\Omega(n) = 1$$

Complexidade de Algoritmo

Cenários possíveis:

- Caso Médio:

Descreve o cenário mais comum, o cenário médio, de número de instruções;

No exemplo da lista, na média os valores procurados estarão no meio da lista. Algumas vezes estarão no inicio, porém outra vezes estarão no fim, isto torna nosso valor médio sendo T(n) = n / 2, utilizando a notação:

$$\theta(n) = \frac{n}{2}$$

Complexidade de Algoritmo

Cenários possíveis:

- Pior Caso:

Descreve o maior número de instruções possíveis de ocorrer para um determinado algoritmo (conforme os dados utilizados);

No exemplo da lista, o valor pesquisado se encontraria no fim e, por tanto, precisaríamos de T(n) = n, com a notação:

$$O(n) = n$$

Complexidade de Algoritmo

- Representações comuns: – O cenário utilizado para representar um algoritmo é o de pior caso;

- Apesar de podermos quantificar as instruções necessárias para cada algoritmo, para o pior caso, o mais comum é arredondarmos para uma das complexidade mais comuns;

$\log_2 n$

n

$n \log_2 n$

n^2

n^3

2^n

3^n

Complexidade de Algoritmo

- Ao medir o número de instruções necessárias para cada algoritmo, devem ser consideradas estruturas, tais como de decisão ou de repetição;

- Em geral, a estratégia de um algoritmo é responsável pela sua eficiência, mas vale lembrar que estruturas de repetição aninhadas são as maiores responsáveis pelo aumento de instruções;

Complexidade de Algoritmo

Complexidade	Tempo para executar N = 100, em um Core i7 980-X Extreme Edition
$\log_2 n$	133 *ns*
n	2 *µs*
$n \log_2 n$	13 *µs*
n^2	200 *µs*
n^3	20 *ms*
2^n	815.104.552.615.888 anos!
3^n	331.389.866.725.830.000 000.000.000.000.000 anos!

Complexidade de Algoritmo

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	

Complexidade de Algoritmo

- Analisar duas soluções para o famoso problema Two Sum, que verifica se existem 2 números cuja soma resulte na entrada específica (alvo) e retorna as posições dos números encontrados:

```go
func twoSum(nums []int, target int) (index1, index2 int) {
	for index1, _ := range nums {
		for index2, _ := range nums {
			if nums[index1]+nums[index2] == target {
				return index1, index2
			}
		}
	}
	return -1, -1
```

Algoritmo em Golang

Complexidade de Algoritmo

- Analisar duas soluções para o famoso problema Two Sum, que verifica se existem 2 números cuja soma resulte na entrada específica (alvo) e retorna as posições dos números encontrados:

Algoritmo em Kotlin

```kotlin
fun twoSum(nums: IntArray, target: Int): Pair<Int, Int> {
    val seen = hashMapOf<Int, Int>()

    nums.forEachIndexed { index, value ->
        if (seen.contains(target - value)) {
            return Pair(seen[target - value]!!, index)
        }
        seen.put(value, index)
    }

    return Pair(-1, -1)
```

Complexidade de Algoritmo

- Rodando as duas soluções com o arrays de até 1000 elementos, o algoritmo em **Go é muito mais rápido que Kotlin**, demorando apenas alguns nanosegundos para encontrar o resultado, enquanto Kotlin demora poucos milisegundos na maioria dos casos.

Será que podemos afirmar que o algoritmo em Go é melhor que o em Kotlin?

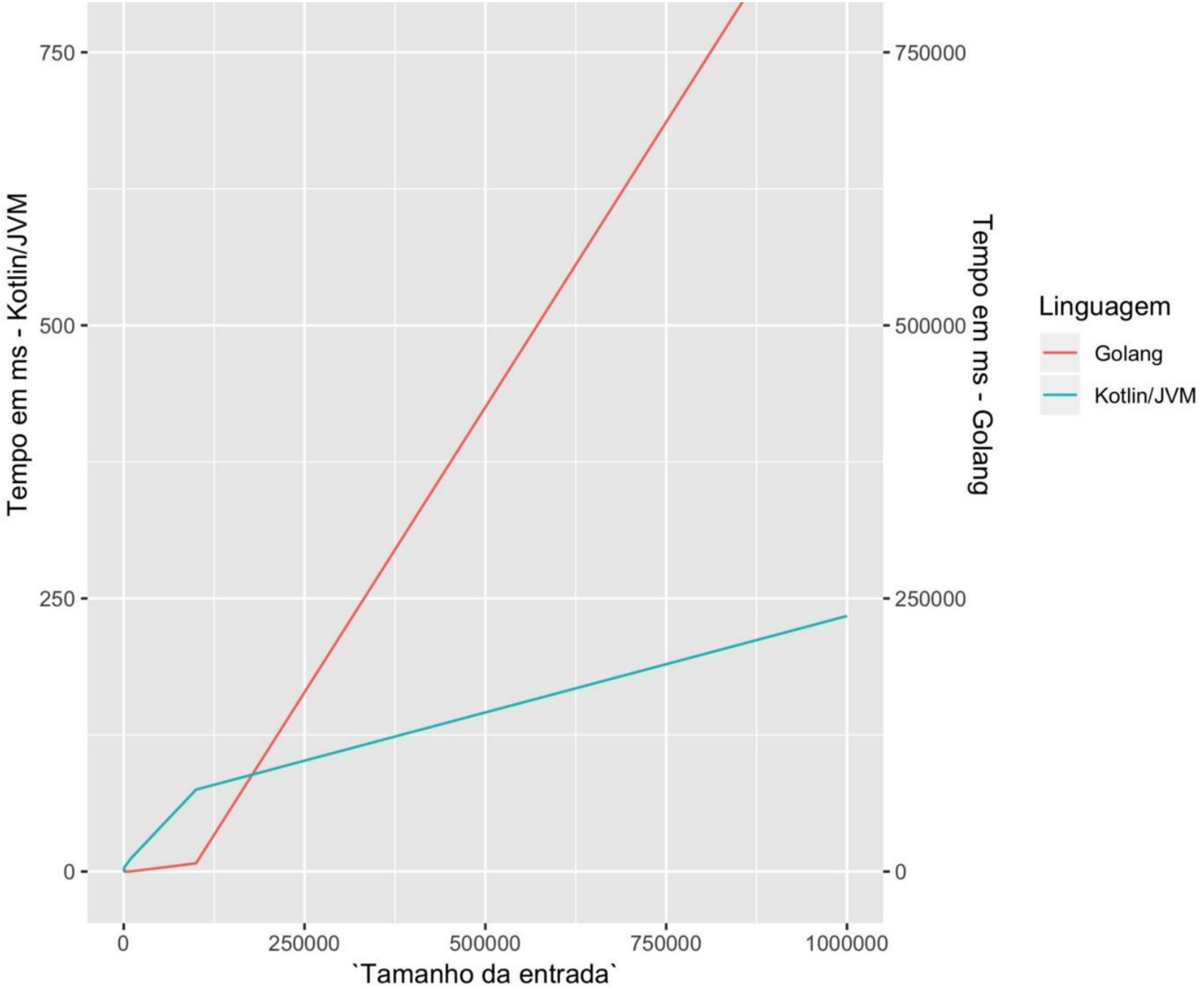
Para isso, precisamos definir o que realmente é um “algoritmo melhor”.

Complexidade de Algoritmo

Para isso, precisamos definir o que realmente é um “algoritmo melhor”.

- Podemos dizer que o melhor algoritmo para resolver um problema é aquele que possui a menor complexidade de tempo e espaço.
- Em outras palavras, é o algoritmo que, conforme a entrada cresce tendendo ao infinito, é aquele que apresenta a menor variação de tempo e memória utilizada para terminar.

Rodando os mesmos algoritmos com entradas variando de 0 a 1 milhão e extraindo o tempo de execução podemos montar o seguinte gráfico:



Complexidade de Algoritmo

- Conforme a entrada cresce, o tempo de execução em Kotlin varia muito pouco, apresentando valores entre 0 e 234 milisegundos.
- Já Go, apresenta tempos de execução entre 0 e 15 minutos com entradas muito grandes.
- Um algoritmo pode ser melhor que outro quando processa poucos dados, porém pode ser muito pior conforme o dado cresce.

Complexidade de Algoritmo

A Análise de complexidade nos permite medir o quão rápido um programa executa suas computações.

Exemplos de computações são:

- Operações de adição e multiplicação;
- Comparações;
- Pesquisa de elementos em um conjunto de dados;
- Determinar o caminho mais curto entre diferentes pontos;
- Ou até verificar a presença de uma expressão regular em uma string.

Complexidade de Algoritmo

O maior elemento de um array de N elementos pode ser encontrado com o seguinte algoritmo:

```
maior := lista[0]


for indice := 0; indice < n; indice++ {
  if lista[indice] > maior {
    maior = lista[indice]
  }
```

FIM!