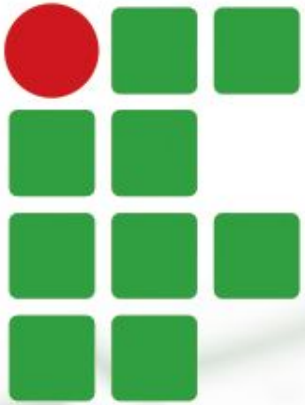


INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Estruturas de Dados 2
- Análise de Algoritmos -

Prof. MSc. Adriano Antunes Prates

Introdução

- Introdução à Análise de Algoritmos
- **O que é relevante investigar sobre um Algoritmo?**

Introdução

- Introdução à Análise de Algoritmos
- **O que é relevante investigar sobre um Algoritmo?**

- *Legibilidade, Simplicidade, Modularidade, Facilidade para Manutenção…*
- *Corretude…*
- ***Eficiência/Desempenho***

Introdução

- Introdução à Análise de Algoritmos
- **Como determinar se um algoritmo é eficiente?**

Introdução

- Introdução à Análise de Algoritmos
- **Como determinar se um algoritmo é eficiente?**

Tempo de Execução?

Introdução

- Introdução à Análise de Algoritmos
- **Como determinar se um algoritmo é eficiente?**

Tempo de Execução?

mas como comparar tempos de execução?

implementar várias soluções possíveis e medir o tempo?

e quanto aos ambientes/recursos heterogêneos?

Introdução

- Introdução à Análise de Algoritmos
- **Como determinar se um algoritmo é eficiente?**

Tempo de Execução?

Problema: A **análise empírica** é muito dependente de **fatores externos ao código**.

P.ex.: Qual o *hardware* disponível?

Qual carga de trabalho no instante da avaliação?

Introdução

- Introdução à Análise de Algoritmos

- **A Análise de um Algoritmo deve se basear em um método simples, padronizado, independente de plataforma, e que seja possível realizá-la sem a obrigatoriedade da sua execução.**

Introdução

- Introdução à Análise de Algoritmos

Analisar um algoritmo significa prever a quantidade de recursos computacionais (processamento, memória, disco, largura de banda, ...) que tal algoritmo necessita consumir para ser executado.

Introdução

- Observe o seguinte Algoritmo…

```
float obter_valor_bruto(float valor, int taxa){
    float imposto;
    int taxa_proporcional;
    if(valor < LIMITE){
        taxa_proporcional = (valor / LIMITE) * taxa;
        imposto = valor * (taxa_proporcional/100);
    } else
        imposto = valor * (taxa/100);
    return valor + imposto;
}
```

Introdução

- Observe o seguinte Algoritmo…

```
float obter_valor_bruto(float valor, int taxa){
    float imposto;
    int taxa_proporcional;
    if(valor < LIMITE){
        taxa_proporcional = (valor / LIMITE) * taxa;
        imposto = valor * (taxa_proporcional/100);
    } else
        imposto = valor * (taxa/100);
    return valor + imposto;
}
```

Observe que o consumo de recursos (processamento e memória) desse algoritmo INDEPENDE da entrada de dados.

Introdução

- Observe o seguinte Algoritmo…

```
float obter_valor_bruto(float valor, int taxa){
    float imposto;
    int taxa_proporcional;
    if(valor < LIMITE){
        taxa_proporcional = (valor / LIMITE) * taxa;
        imposto = valor * (taxa_proporcional/100);
    } else
        imposto = valor * (taxa/100);
    return valor + imposto;
}
```

Olhares mais atentos dirão: MAS A DEPENDER DA ENTRADA HAVERÁ UMA INSTRUÇÃO A MAIS…

Análise Assintótica

- Entretanto, essa variação é **insignificante** para a **Análise Assintótica de Algoritmos.**

Ao ver uma expressão como **n+10**, **50*n** ou **n^2+8**, a maioria das pessoas pensa automaticamente em valores pequenos de **n**.

A **Análise Assintótica** faz exatamente o contrário: *ignora os valores pequenos* e concentra-se apenas nos valores *enormes* de **n**.

Para valores enormes de **n**, as funções **$2n^2$, $(3/2)n^2$, $9999*n^2$, $n^2/1000$, n^2+100n, CRESCEM TODAS COM A MESMA VELOCIDADE, E PORTANTO, SÃO TODAS EQUIVALENTES EM COMPLEXIDADE**
ou seja, podemos considerar que todas são funções n^2

Análise Assintótica

- A **Análise Assintótica de Algoritmos** é uma abstração que busca avaliar o comportamento de um algoritmo em relação ao tratamento de grandes **volume de dados**.

Premissas Básicas para a Análise Assintótica:

“O custo de operações primitivas é constante”

“Sempre considera-se o pior caso”

Análise Assintótica

- Observe agora um outro exemplo…

```
int busca(int array[], int tamanho, int valor){
   for(int i=0; i < tamanho; i++)
      if(array[i] == valor)
         return i;
   return -1;
}
```

Neste exemplo, a entrada de dados impacta diretamente na eficiência do algoritmo?

- A **Análise Assintótica** é uma abstração que busca avaliar o comportamento de um algoritmo em relação a grandes **volume de dados**.

Análise Assintótica

```
for(int i=0; i<N; i++){
    print(i);
}
```

Poderíamos considerar que o tempo de execução do código acima seja...

```
T(N) =
    N*(tempo gasto por uma comparação entre i e N) +
    N*(tempo gasto para incrementar i) +
    N*(tempo gasto por um print)
```

Ou, como operações primitivas são constantes...

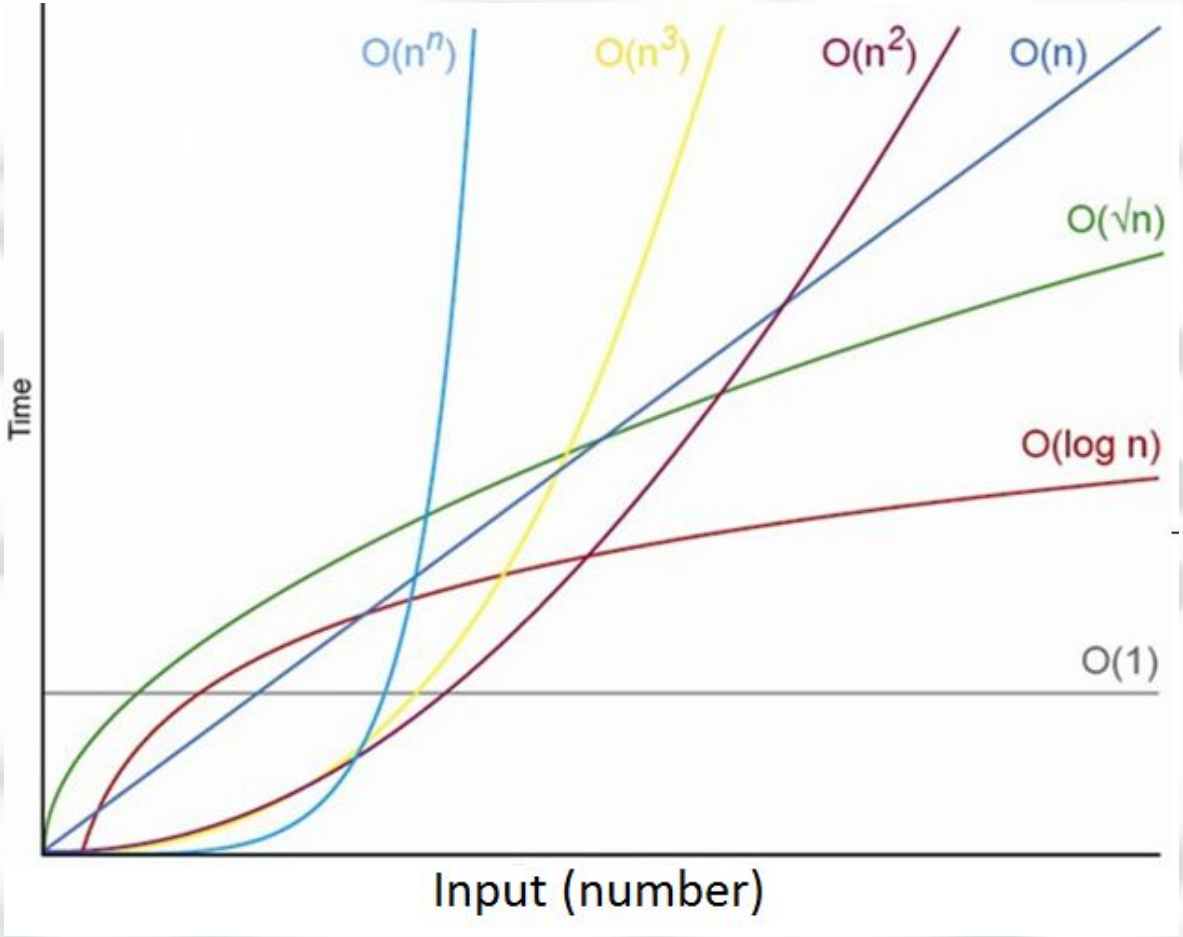
```
T(N) = 3*N
```

Ou, simplesmente...

`T(N) = N` ⇒ `O(N)` ⇒ `Ordem Linear`

Análise Assintótica

- **Análise Assintótica e Classes BIG-O**



Notação “BIG O”

- A notação **Big O** é o método mais comum de se descrever o nível de complexidade - ou eficiência/velocidade - de um algoritmo.
- Esta notação leva em consideração a relação **Entrada de Dados** vs. **Número de Operações** que um algoritmo deve executar para atingir um determinado objetivo.
- Vamos observar um exemplo simples e direto…

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Notação "BIG O"

- Considere que: o tempo de consulta de um registro em um Array leva 1 ms. Você precisa encontrar um registro específico…

Número de Elementos no Array	Tempo Máximo	
	Algoritmo: Busca Sequencial	Algoritmo: Busca Binária
10	10 ms	4 ms
100	100 ms	7 ms
10.000	10 seg	14 ms

Notação "BIG O"

- Considere que: o tempo de consulta de um registro em um Array leva 1 ms. Você precisa encontrar um registro específico…

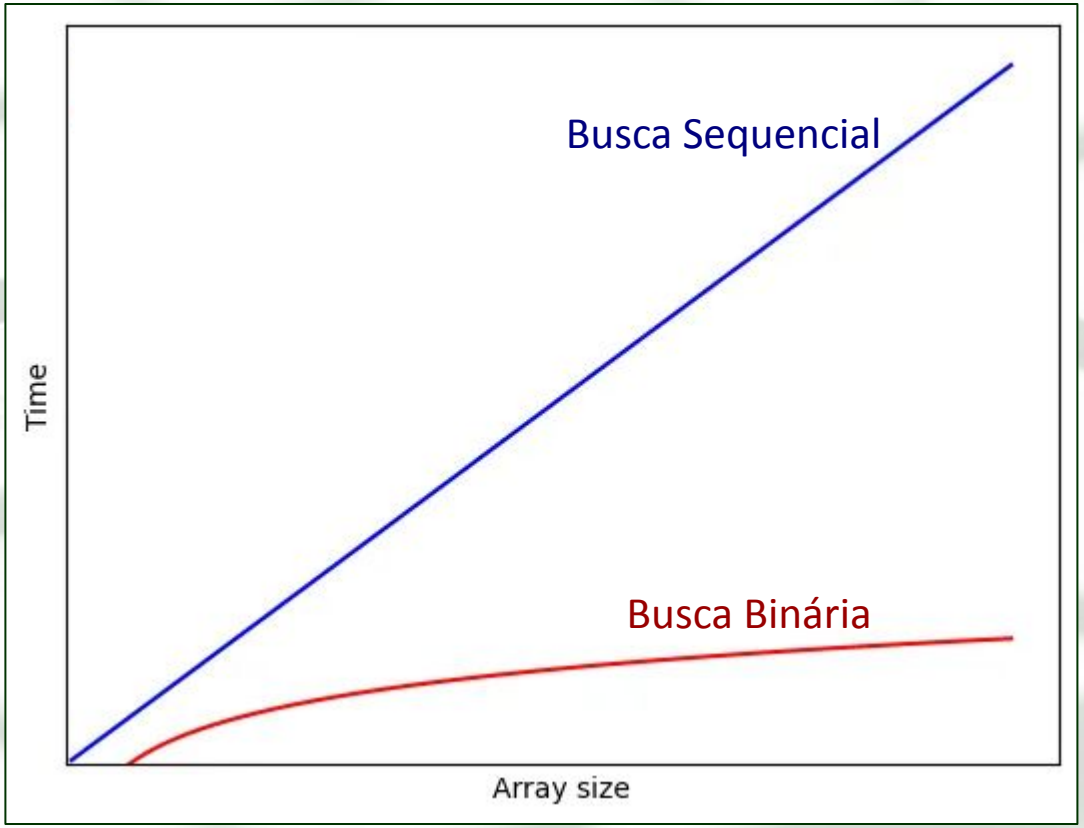
Número de Elementos no Array	Tempo Máximo	
	Algoritmo: Busca Sequencial	Algoritmo: Busca Binária
10	10 ms	4 ms
100	100 ms	7 ms
10.000	10 seg	14 ms
1.000.000.000	**11 DIAS**	**30 MS**

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Notação "BIG O"

- Fica nítido que os tempos de execução **não tem a mesma taxa de crescimento**.

Conforme o número de elementos cresce, o tempo da busca binária **aumenta minimamente**, enquanto o tempo de execução da busca sequencial **aumenta proporcionalmente**.

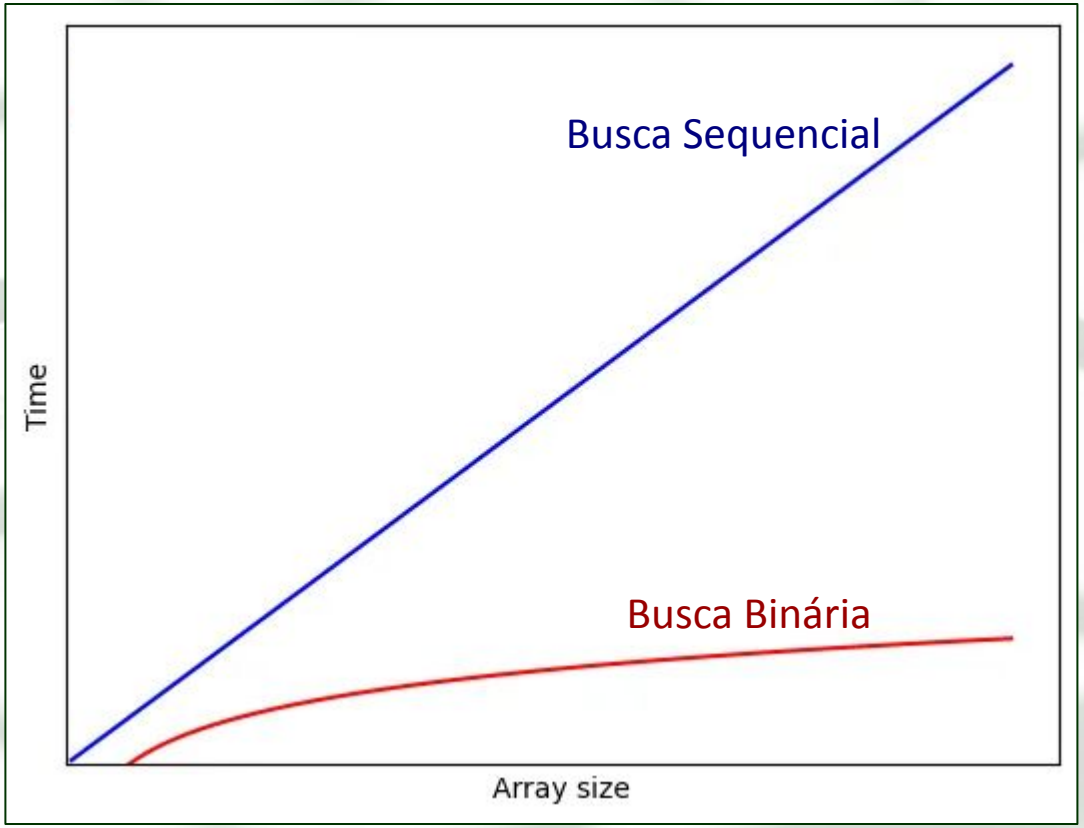


INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Notação "BIG O"

- Fica nítido que os tempos de execução **não tem a mesma taxa de crescimento**.

A notação Big O nos faz compreender como funciona a **escalabilidade de um algoritmo**, ou seja, como **o tamanho da entrada de dados pode impactar o tempo de resposta de um código**.

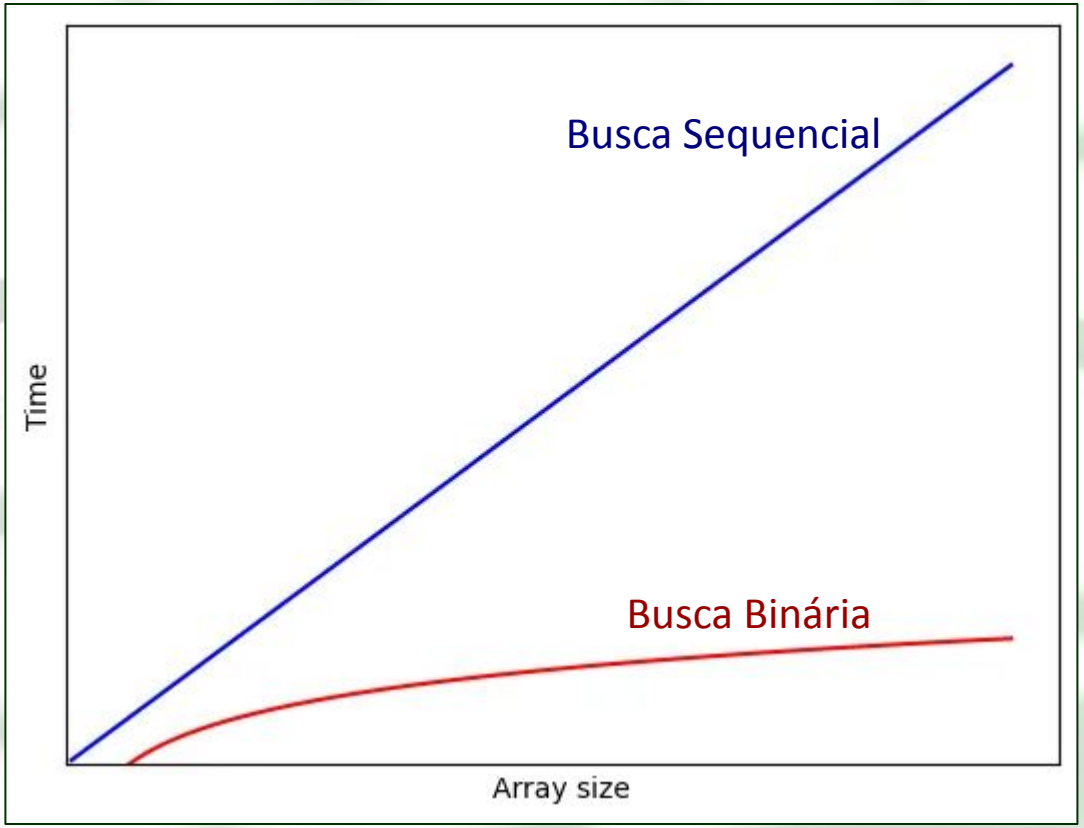


INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Notação "BIG O"

- Fica nítido que os tempos de execução **não tem a mesma taxa de crescimento**.

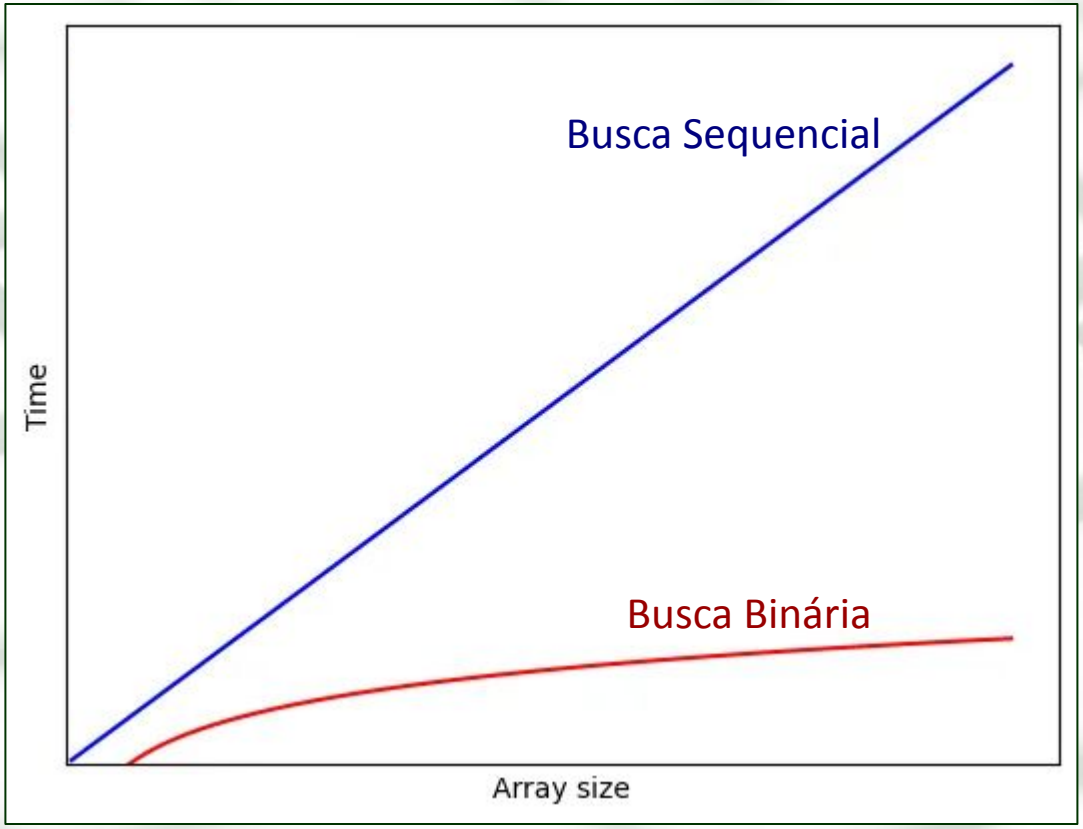
No "**mundo ideal**" a performance de um algoritmo deve (parecer) ser a mesma, **tanto para uma entrada de dados pequena, quanto para uma entrada de dados gigantesca.**



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Notação "BIG O"

- E qual é o segredo da "mágica" aqui???
- Qual a diferença entre os dois algoritmos???



Ordem Constante

- Algoritmos de **Ordem Constante**

O(1)

- São algoritmos em que o número de operações (tempo de execução) **independe** da quantidade de elementos.
- P.Ex.:

```
void multi(int array[], int indice, int valor){
    array[indice] *= valor;
}
```

Ordem Constante

- Algoritmos de **Ordem Constante**

O(1)

- São algoritmos em que o número de operações (tempo de e
elementos.
- P.Ex.:

Essa função irá executar uma única operação, independentemente se o array for de tamanho 10 ou 1.000.000.000

```
void multi(int array[], int indice, int valor){
    array[indice] *= valor;
}
```

Ordem Constante

- E se fosse uma função assim...

```
void troca(int array[], int indice){
    int temp = array[indice];
    array[indice] = array[indice+1];
    array[indice+1] = temp;
}
```

Ordem Constante

- E se fosse uma função assim...

```
void troca(int array[], int indice){
    int temp = array[indice];
    array[indice] = array[indice+1];
    array[indice+1] = temp;
}
```

O(3) ou O(1) ???

Ordem Constante

- E se fosse uma função assim...

```
void troca(int array[], int indice){
    int temp = array[indice];
    array[indice] = array[indice+1];
    array[indice+1] = temp;
}
```

O(3) ou O(1) ???

- Para notação Big O ainda seria O(1), pois embora faça mais operações, o tempo de execução é **CONSTANTE**, independente do tamanho do Array.

Ordem Constante

- E se fosse uma função assim…

```
void troca(int array[], int indice){
    int temp = array[indice];
    array[indice] = array[indice+1];
    array[indice+1] = temp;
}
```

Para Big O, não nos preocupamos se o código é O(1), O(2), O(23), O(647653), etc…

Arredonda-se para O(1), pois **a operação é uma linha plana em termos de escalabilidade**, e levaria a mesma quantidade de tempo independente do tamanho da entrada de dados.

Ordem Constante

- E se fosse uma função assim...

```
void troca(int array[], int indice){
    int temp = array[indice];
    array[indice] = array[indice+1];
    array[indice+1] = temp;
}
```

Para Big O, não nos preocupamos se o código é O(1), O(2),

Não queremos dizer que **O(1)** executará na mesma quantidade de tempo de **O(647653)** mas sim que **AMBOS ESTÃO NA MESMA CLASSE DE COMPLEXIDADE**.

de tempo independente do tamanho da entrada de dados.

Ordem Linear

- Algoritmos de **Ordem Linear**

O(n)

- São algoritmos em que o número de operações **aumenta proporcionalmente** à quantidade de elementos...

```
void multi(int array[], int cont, int valor){
    for(int i=0; i<cont; i++)
        array[i] *= valor;
}
```

Ordem Linear

- Algoritmo de busca sequencial…
  - **Constante ou Linear?**

```
int busca(int array[], int cont, int alvo){
    for(int i=0; i<cont; i++)
        if(array[i] == alvo)
            return i;
    return -1;
}
```

Ordem Linear

- Algoritmo de busca sequencial…
  - **Constante ou Linear?**

```
int busca(int array[], int cont, int alvo){
    for(int i=0; i<cont; i++)
        if(array[i] == alvo)
            return i;
    return -1;
}
```

A notação Big O **sempre** considera o **PIOR** caso de execução.

Entretanto, existem outras notações (Omega Ω, Theta Θ) que consideram o melhor caso e caso médio, respectivamente.

Ordem Quadrática

- Algoritmos de **Ordem Quadrática**

$$O(n^2)$$

- São algoritmos em que para N entrada de dados, precisaremos realizar N * N operações.

- Algum exemplo em mente???

Ordem Quadrática

- Bubble Sort | Selection Sort | Insertion Sort

$O(n^2)$

```
void ordena(int array[], int cont){
   for(int i=0; i<cont; i++)
      for(int j=0; j<cont-i-1; j++)
         if(array[j] > array[j+1])
            troca(array,j);
}
```

Ordem Quadrática

- Bubble Sort | Selection Sort | Insertion Sort

$O(n^2)$

```
void ordena(int array[], int cont){
  for(int i=0; i<cont; i++)
    for(int j=0; j<cont-i-1; j++)
      if(array[j] > array[j+1])
        troca(array,j);
}
```

repete n vezes

repete (n-i) vezes

Problema Prático

- Imagine o seguinte problema…

Leia vários números aleatórios no intervalo entre 1 e 100.
Interrompa a leitura quando encontrar o primeiro valor repetido.

Problema Prático

- Imagine o seguinte problema…

Leia vários números aleatórios no intervalo entre 1 e 100.
Interrompa a leitura quando encontrar o primeiro valor repetido.

- O algoritmo que você imaginou é CONSTANTE, LINEAR ou QUADRÁTICO?

Problema Prático

- Imagine o seguinte problema…

Leia vários números aleatórios no intervalo entre 1 e 100.
Interrompa a leitura quando encontrar o primeiro valor repetido.

- O algoritmo que você imaginou é CONSTANTE, LINEAR ou QUADRÁTICO?

- É possível uma solução **CONSTANTE**?

Mudando de Assunto…

- Era uma vez um rei indiano que queria recompensar um homem sábio por seu excelente trabalho…

eis que pediu…

"Como gosto muito de jogar Xadrez, quero apenas uma quantidade de trigo equivalente a um tabuleiro de xadrez…

Quero 1 grão de trigo para a primeira casa do tabuleiro, depois 2 na segunda, 4 na terceira, e assim sucessivamente até preencher todo o tabuleiro."

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Mudando de Assunto…

O rei inocente concordou sem hesitar, pensando até que seria muito simples atender ao pedido…

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Mudando de Assunto…

- Resultado da Brincadeira…

$1{,}844674407 \times 10^{19}$ grãos de trigo
que é aproximadamente…
18.000.000.000.000.000.000 grãos.

Se um grão pesa 0,01 grama, temos:
180.000.000.000 TONELADAS, ou
180 BILHÕES DE TONELADAS DE TRIGO

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Mudando de Assunto...

- Quantas vezes você seria capaz de dobrar uma folha de papel ao meio?

Mudando de Assunto…

- Havendo papel (e energia) suficiente…
- Dobrando 13 vezes você quebraria o **recorde mundial** (atualmente é 12) e entraria guiness book.

Mudando de Assunto…

- Havendo papel (e energia) suficiente…

- Dobrando 13 vezes você quebraria o recorde mundial (atualmente é 12) e entraria guiness book.
- Com 30 dobras você já poderia subir nele e **chegar ao espaço** (haveria 100 KM de altura).

Mudando de Assunto…

- Havendo papel (e energia) suficiente…

- Dobrando 13 vezes você quebraria o recorde mundial (atualmente é 12) e entraria guiness book.
- Com 30 dobras você já poderia subir nele e chegar ao espaço (haveria 100 KM de altura).
- Com 42 dobras você **chegaria à Lua**, e 51 ao **Sol**.

Mudando de Assunto…

- Havendo papel (e energia) suficiente…

- Dobrando 13 vezes você quebraria o recorde mundial (atualmente é 12) e entraria guiness book.
- Com 30 dobras você já poderia subir nele e chegar ao espaço (haveria 100 KM de altura).
- Com 42 dobras você chegaria à Lua, e 51 ao Sol.
- Com 103 dobras teria uma espessura do **tamanho do universo observável** (93 bilhões de anos-luz)

Fonte

Ordem Exponencial

- O que isso tem haver com a matéria?
- TUDO! Dobrar papel trata-se de um exemplo de problema de **Ordem Exponencial**.
- Algoritmos de **Ordem Exponencial**

$$O(2^n)$$

Ordem Exponencial

- Algoritmos de **Ordem Exponencial** só perdem para os de **Ordem Fatorial** em relação à complexidade e custo (número de operações e tempo de execução envolvidos).

- Um exemplo de Algoritmo de Complexidade Exponencial são os de **Brute Force Attack**.

- *P.Ex.:* Uma senha forte de 12 caracteres levaria aproximadamente 8 milhões de anos para ser quebrada, mesmo nos supercomputadores.

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Ordem Exponencial

Password Length	Numerical 0-9	Upper & Lower case a-Z	Numerical Upper & Lower case 0-9 a-Z	Numerical Upper & Lower case Special characters 0-9 a-Z %$
1	instantly	instantly	instantly	instantly
2	instantly	instantly	instantly	instantly
3	instantly	instantly	instantly	instantly
4	instantly	instantly	instantly	instantly
5	instantly	instantly	instantly	instantly
6	instantly	instantly	instantly	20 sec
7	instantly	2 sec	6 sec	49 min
8	instantly	1 min	6 min	5 days
9	instantly	1 hr	6 hr	2 years
10	instantly	3 days	15 days	330 years
11	instantly	138 days	3 years	50k years
12	2 sec	20 years	162 years	8m years
13	16 sec	1k years	10k years	1bn years
14	3 min	53k years	622k years	176bn years
15	26 min	3m years	39m years	27tn years
16	4 hr	143m years	2bn years	4qdn years
17	2 days	7bn years	148bn years	619qdn years
18	18 days	388bn years	9tn years	94qtn years
19	183 days	20tn years	570tn years	14sxn years
20	5 years	1qdn years	35qdn years	2sptn years

Ordem Logarítmica

- Saindo de um extremo (Algoritmos de maior ordem de complexidade) e indo para o extremo oposto, temos os **Algoritmos de Ordem Logarítmica**.

- Algoritmos de **Ordem Logarítmica**

$$O(\log_2 n)$$

ou simplesmente . . .

$$O(\log n)$$

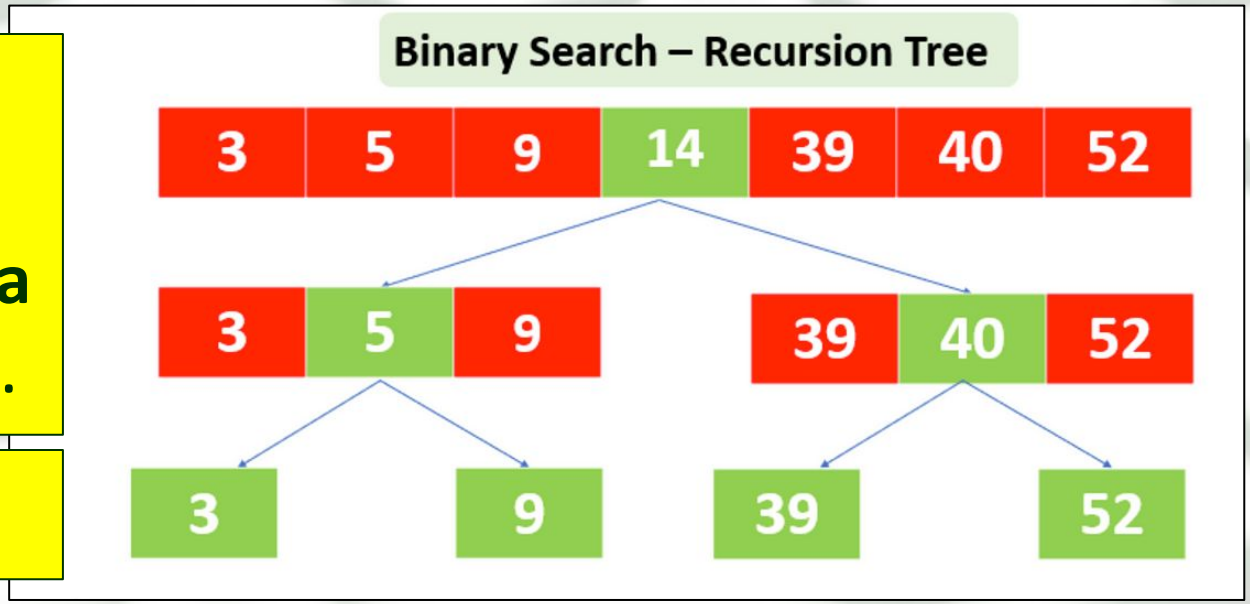
INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Ordem Logarítmica

- Algoritmos de Ordem Logarítmica caracterizam-se pela delimitação da entrada de dados (geralmente 50%) a cada iteração realizada…

Exemplo clássico de algoritmo de **ordem logarítmica** é a **Busca Binária**.

Log_2 7 == 3

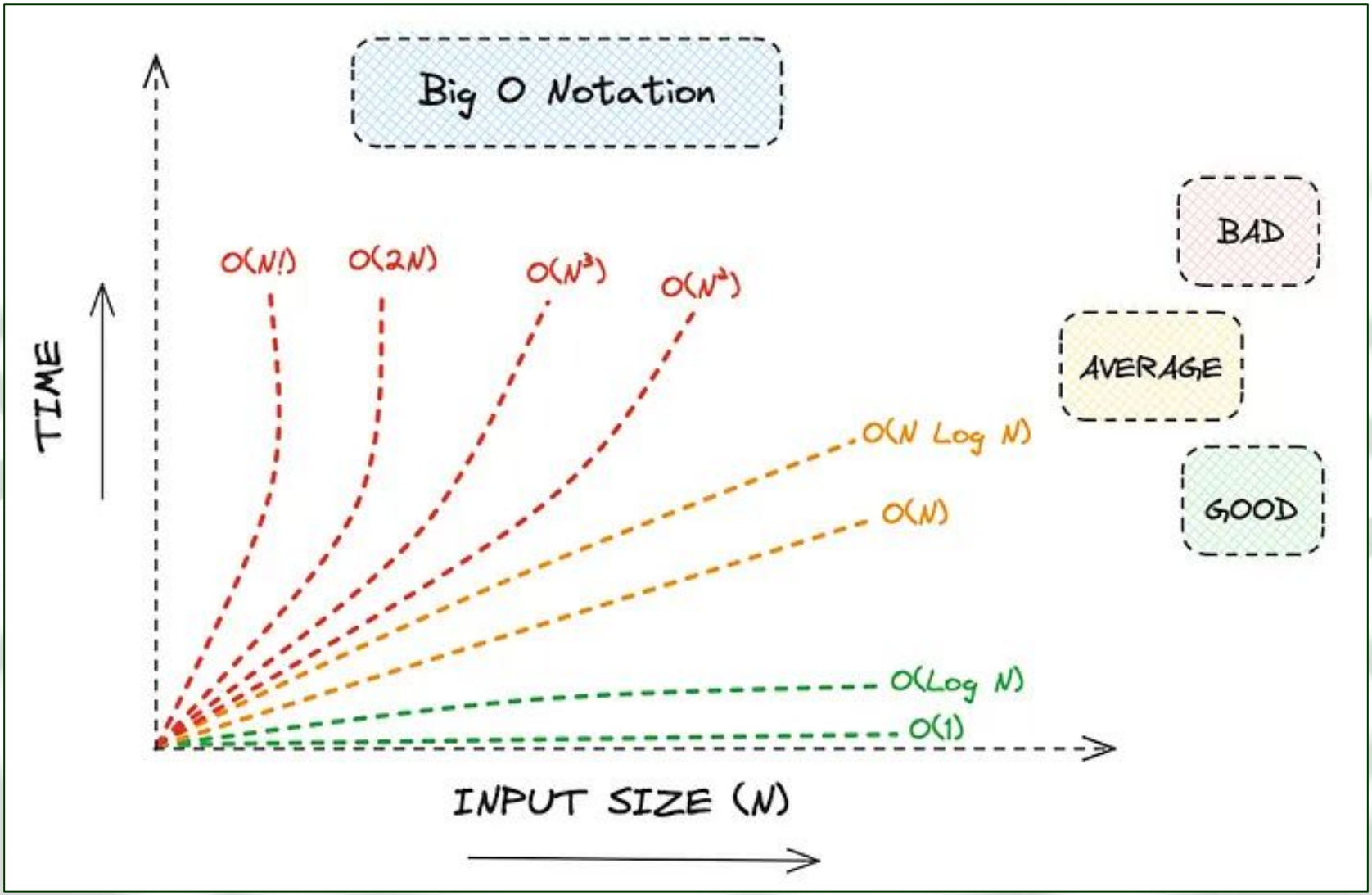


Principais Classificações

Ordem	Classificação	Exemplo de Algoritmo
O(1)	Constante	Acesso direto a elementos
O(n)	Linear	Busca Sequencial
$O(n^2)$	Quadrático	Selection Sort
O(log n)	Logarítmico	Busca Binária
O(n log n)	Linearítmica	Merge Sort
$O(2^n)$	Exponencial	Brute Force
O(n!)	Fatorial	Caixeiro Viajante

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Principais Classificações



Complexidades

- Qual é a complexidade dos algoritmos clássicos de ordenação?
  - Bubble, Insertion e Selection?

- Seria possível criar um algoritmo de ordenação **tão simples quanto os clássicos**, mas com uma eficiência melhor?

Complexidades

- Analise o Funcionamento do Método de Ordenação **Counting Sort**

- LINK

- Qual é a **ordem de complexidade** deste método?

Complexidades

- Qual é a **ordem de complexidade** deste método?
  - 1 Laço de repetição para contagem dos elementos (N => Linear)
  - 1 Laço de repetição para espalhar as K chaves contadas (K => Linear)

- Perceba… 02 laços de repetição simples, não-aninhados, diferentemente dos outros algoritmos clássicos.

O(n+k), ou simplesmente O(n)

Complexidades

- Qual é a **ordem de complexidade** deste método?
  - 1 Laço de repetição para contagem dos elementos (N => Linear)
  - 1 Laço de repetição para espalhar as K chaves contadas (K => Linear)

- Perceba… 02 laços de repetição simples, não-aninhados, diferentemente dos outros algoritmos clássicos.

Qual é o porém entretanto?

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Complexidade de Espaço

- Embora a complexidade de operações (tempo) seja mais eficiente (Linear), a complexidade de espaço no **Counting Sort** é maior.
- A ordenação não é *in-place*, porque exige espaço de memória extra, proporcional ao tamanho de k.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Complexidade de Espaço

- Embora a complexidade de operações (tempo) seja mais eficiente (Linear), a complexidade de espaço no **Counting Sort** é maior.
- A ordenação não é *in-place*, porque exige espaço de memória extra, proporcional ao tamanho de k.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)

?

INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

Complexidade de Espaço

- Embora a complexidade de operações (tempo) seja mais eficiente (Linear), a complexidade de espaço no **Counting Sort** é maior.
- A ordenação não é *in-place*, porque exige espaço de memória extra, proporcional ao tamanho de k.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)

Memória adicional requerida INDEPENDE do tamanho da entrada de dados.

Complexidades

- **Complexidade de Tempo (Operações)**
  - **Número de operações** que um algoritmo necessita para completar seu objetivo à medida que a entrada de dados aumenta.

- **Complexidade de Espaço**
  - **Quantidade de memória** utilizada por um algoritmo durante sua execução à medida que a entrada de dados aumenta.

Referências

- Big-O Cheat Sheet
- Iniciando a Notação Big O
- O que é Notação Big O