

# ESTRUTURA DE DADOS I

## Aula 01

Prof. Sérgio Luis Antonello

FHO - Fundação Hermínio Ometto

17/02/2025

# Cronograma do Plano de Ensino



- 17/02 - Recursividade; Complexidade de tempo; Notação Big-Oh.
- 24/02 - Métodos de ordenação: Bubble sort; Insert sort; Select sort.
- 10/03 - Métodos de ordenação: Quick sort; Merge sort.
- 17/03 - Métodos de ordenação: Shell sort; Radix sort.
- 24/03 - Métodos de pesquisa: Sequencial; Binária.
- 31/03 - Métodos de pesquisa: Hashing.
- 07/04 – Desenvolvimento do trabalho A1.
- **14/04 - Prova 1**

# Sumário

---

- Primeiro momento
  - Revisão de conceitos importantes
- Segundo momento
  - Recursividade
  - Complexidade de tempo
- Terceiro momento
  - Síntese



# ***Primeiro momento***

# 1. Algoritmos x Estruturas de Dados

---




## ■ Algoritmo

- Sequência de ações executáveis para a solução de um determinado tipo de problema
- De maneira geral os algoritmos trabalham sobre as estruturas de dados

## ■ Estruturas de Dados

- Conjunto de dados que representa uma situação real
- Consiste na abstração da realidade

# 1. Algoritmos x Estruturas de Dados

mundo real	dados de interesse	ESTRUTURA de armazenamento	possíveis OPERAÇÕES
 pessoa	<ul style="list-style-type: none"><li>a idade da pessoa</li></ul>	<ul style="list-style-type: none"><li>tipo inteiro</li></ul>	<ul style="list-style-type: none"><li>nasce (<math>i \leftarrow 0</math>)</li><li>aniversário (<math>i \leftarrow i + 1</math>)</li></ul>
 cadastro de funcionários	<ul style="list-style-type: none"><li>o nome, cargo e o salário de cada funcionário</li></ul>	<ul style="list-style-type: none"><li>tipo lista ordenada</li></ul>	<ul style="list-style-type: none"><li>entra na lista</li><li>sai da lista</li><li>altera o cargo</li><li>altera o salário</li></ul>
 fila de espera	<ul style="list-style-type: none"><li>nome de cada pessoa e sua posição na fila</li></ul>	<ul style="list-style-type: none"><li>tipo fila</li></ul>	<ul style="list-style-type: none"><li>sai da fila (o primeiro)</li><li>entra na fila (no fim)</li></ul>

## 2. Linguagem C: tipos básicos de dados

---

- Conhecidos como tipos primitivos
- Devem ser utilizados de acordo com o tipo de armazenamento e processamento que se fizer necessário
- Exemplo: dados de um funcionário
  - Nome: José Antonio Cruz Folher
  - Idade: 30
  - Sexo: M
  - Salário: 2.500,00

## 2. Linguagem C: tipos básicos de dados

---

### ■ Tipos

- int
- float
- double
- char

### ■ void

### ■ Modificadores

- signed
- unsigned
- long
- short



### 3. Linguagem C: tipos e modificadores

hardware 64 bits (a depender do compilador)

TIPO	Bytes	Formato para leitura com scanf	Intervalo
char	1	%c	-127 a 127
unsigned char	1	%c	0 a 255
int	4	%i	-2.147.483.648 a 2.147.483.647
unsigned int	4	%u	0 a 4.294.967.295
short int	2	%hi	-32.768 a 32.767
unsigned short int	2	%hu	0 a 65.535
long int	4	%li	-2.147.483.648 a 2.147.483.647
long long int	8	%lli	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
unsigned long int	4	%lu	0 a 4.294.967.295
float	4	%f	$3.4 \times 10^{-38}$ a $3.4 \times 10^{38}$
double	8	%lf	$1.7 \times 10^{-308}$ a $1.7 \times 10^{308}$
long double	10	%Lf	$3.4 \times 10^{-4932}$ a $3.4 \times 10^{4932}$

# 3. Vetor

---

- Estrutura homogênea

- O que é?

É um conjunto de valores de um mesmo tipo.

- Como os elementos ficam alocados na memória?

Esses valores ficam alocados consecutivamente na memória.

- Como os elementos podem ser acessados?

Eles podem ser acessados a partir da sua posição, identificada como índice.

# 3. Vetor

- Um elemento pode ser usado pelo programa como uma variável qualquer?

Sim, um elemento do vetor pode ser usado no programa como qualquer outra variável.

- Qual o índice do primeiro elemento do vetor?

O primeiro elemento do vetor tem índice zero.

- Qual o índice do último elemento do vetor?

O último elemento tem índice  $n-1$ .

- Um índice pode ser uma variável?

Um índice pode ser uma constante ou uma variável do tipo inteiro.

### 3. Vetor: exemplos de declaração

---

- `int V[10];`
- `char nome[30];`
- `float salario[10];`

```
for (i=0; i<10; i++) {  
    scanf("%d", &V[i]);  
}
```

# 4. Matriz

---

- Estrutura homogênea

- O que é?

É um conjunto de valores de um mesmo tipo.

- Como os elementos ficam alocados na memória?

Esses valores ficam alocados consecutivamente na memória.

- Como os elementos podem ser acessados?

Eles podem ser acessados a partir da sua posição, identificada por índices.

A quantidade de índices depende da dimensão da matriz.

## 4. Matriz: exemplos de declaração

- `int tabela [2][3];`
- `float VendasPorVendedor[10][31];`
- `int tabela [2][3] = {{1,2,3}, {4,5,6}};`

1	2	3
4	5	6

## 4. Matriz: exemplos de declaração

```
#include <stdio.h>
```

```
void main (void) {
```

```
    int linha, coluna;
```

```
    float Mat[3][5] = { {1.0, 2.0, 3.0, 4.0, 5.0},  
                        {6.0, 7.0, 8.0, 9.0, 10.0},  
                        {11.0, 12.0, 13.0, 14.0, 15.0} };
```

```
    for (linha=0; linha < 3; linha++)
```

```
        for (coluna=0; coluna < 5; coluna++)
```

```
            printf("Matriz[%d][%d] = %f\n", linha, coluna,  
                  Mat[linha][coluna]);
```

```
}
```

# 5. Funções

---

- São trechos de código fonte, com finalidades específicas
- Modularização - possibilita estruturar logicamente o código fonte em blocos, o que facilita seu entendimento
- Permite o reaproveitamento de código
- Evita que um trecho de código que seja repetido várias vezes dentro de um mesmo programa
- Agilidade na manutenção do código



## 5. Funções: declaração e chamada

---

- Na declaração, além do nome, devem-se estabelecer os parâmetros de comunicação, através de parâmetros que ela recebe e do conteúdo de retorno.
- Variáveis declaradas dentro de uma função apresentam escopo local.
- Quando um programa precisa fazer uso de uma função, basta ser incluída em seu código uma chamada para a função desejada, respeitando a ordem e os tipos de parâmetros estabelecidos para a comunicação.

## 5. Funções: declaração e chamada

---

- O código da função só é executado quando ela é invocada (chamada) por “alguém”.
- Sempre que uma função é invocada, o programa que a invoca é “suspenso” temporariamente.
- Uma vez terminada a execução da função, o controle de execução volta ao local em que ela foi invocada.
- O programa que invoca pode enviar **argumentos** para a função que os recebe no formato de **parâmetros**.

# 5. Funções: passagem de parâmetro

---

## ■ Passagem por valor

- “Uma cópia” do conteúdo do argumento é encaminhado para a função.
- Se a função alterar o valor recebido (da cópia), o valor da variável original não sofre alteração.

## ■ Passagem por referência.

- O que é encaminhado para a função é o endereço de memória do argumento.
- A função chamada recebe o parâmetro (endereço de memória) em um ponteiro.
- Assim, o parâmetro recebido e o argumento enviado referenciam o mesmo endereço de memória.

## 5. Funções: exemplo

```
int FCalc(int x1, int *x2) {  
    x1 = x1 + 40;  
    *x2 = *x2 - 5;  
    return x1;  
}
```

```
int main() {  
    int v1, v2, r;  
    v1= 10;  
    v2= 20;  
  
    r = FCalc(v1, &v2);  
    printf("Valor1: %d  Valor2: %d  Ret: %d", v1, v2, r);  
}
```



# ***Segundo momento***



***Recursividade***

***e***

***Complexidade  
de Tempo***

# Recursividade



## 6. Recursividade



- Um programa recursivo é um programa que chama a si mesmo, direta ou indiretamente.
- Este conceito possibilita a definição de conjuntos infinitos com comandos finitos.
- A ideia básica consiste em diminuir sucessivamente o problema em um problema menor ou mais simples, até que a simplicidade permita resolvê-lo de forma direta, sem recorrer a si mesmo.



## 6. Recursividade: Implementação

---

- Consiste em dentro do corpo de uma função, chamar novamente a própria função.
  - recursão direta: a função A chama a própria função A
  - recursão indireta: a função A chama uma função B que, por sua vez, chama a A

## 6. Recursividade: Componentes

---

- Condição de parada, quando a parte do problema pode ser resolvida diretamente, sem chamar de novo a função recursiva.
  - Sem a condição de parada pode ocorrer estouro da capacidade da pilha
- Comandos e expressões necessárias para resolver o problema ...
- Chamada recursiva (para resolver parte menor do problema)

## 6. Recursividade: Vantagens x Desvantagens

---

- Um programa recursivo é mais elegante e menor que a sua versão iterativa, além de exibir com maior clareza o processo utilizado, desde que o problema ou os dados sejam naturalmente definidos através de recorrência.
- Por outro lado, um programa recursivo exige mais espaço de memória e é, na grande maioria dos casos, mais lento do que a versão iterativa.

## 6. Recursividade: Característica

---

- Usa-se uma pilha para armazenar cada chamada recursiva
- Cada vez que uma função é chamada de forma recursiva, é criada e armazenada uma cópia dos seus parâmetros de tal maneira que não se perde os valores dos parâmetros das chamadas anteriores.
- Em cada instância da função, só são diretamente acessíveis os parâmetros criados para esta instância, não sendo diretamente acessíveis os parâmetros de outras instâncias.

## 6. Recursividade: Exemplo Fatorial ( $n!$ )

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$5! = 5 * 4 * 3 * 2 * 1$$

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

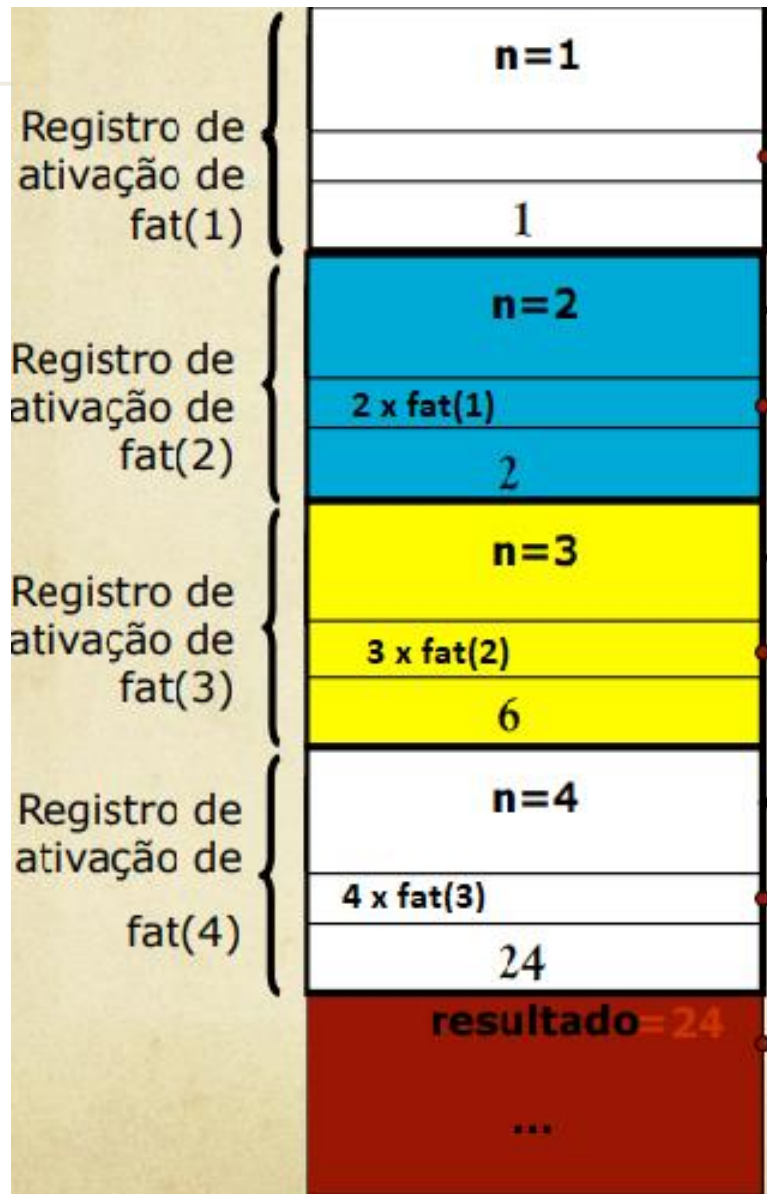
$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

- Cada caso é reduzido a um caso mais simples até chegarmos ao caso  $0!$ , que é definido imediatamente como 1 (ponto de parada).

## 6. Recursividade: Exemplo Fatorial ( $n!$ )



### ■ Recursividade

$$4! = 4 * 3 * 2 * 1$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1$$

## 6. Recursividade: Exemplo Fatorial ( $n!$ )

```
int RecursivaFatorial (int n) {  
    if (n == 0)  
        return (1);        // Condição de parada  
    else  
        return (n * RecursivaFatorial (n-1));  
}
```

```
int NaoRecursivaFatorial (int n) {  
    int i;  
    float result;  
    result = 1;  
    for (i=2; i <= n; i++) {  
        result = result * i;  
    }  
    return (result);  
}
```

## 6. Recursividade: Exemplo Fibonacci

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \quad \text{para } n \geq 2$$

$$\text{fib}(5) = \text{fib}(4) + \text{fib}(3)$$

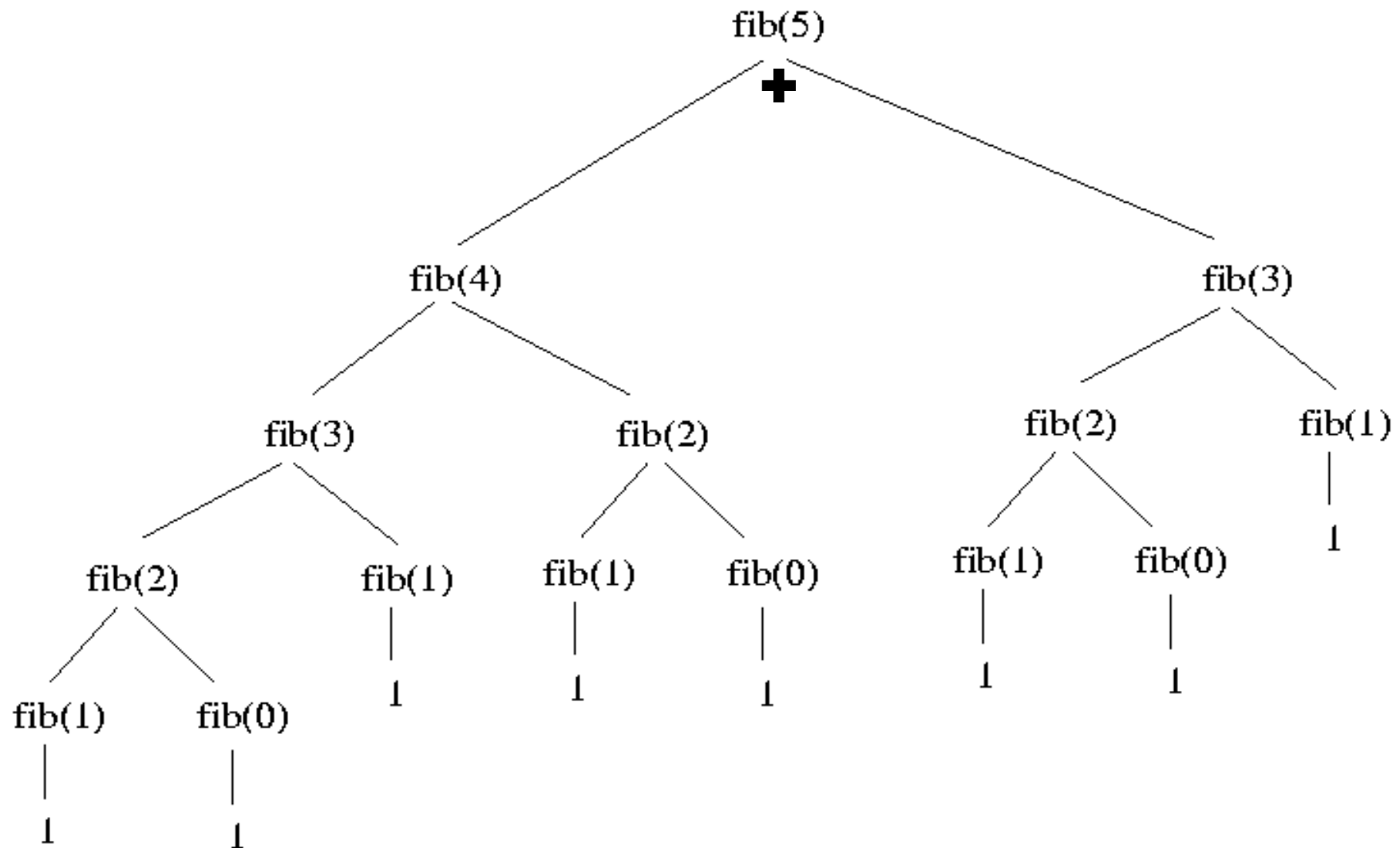
$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$$

...



## 6. Recursividade: Exemplo Fibonacci



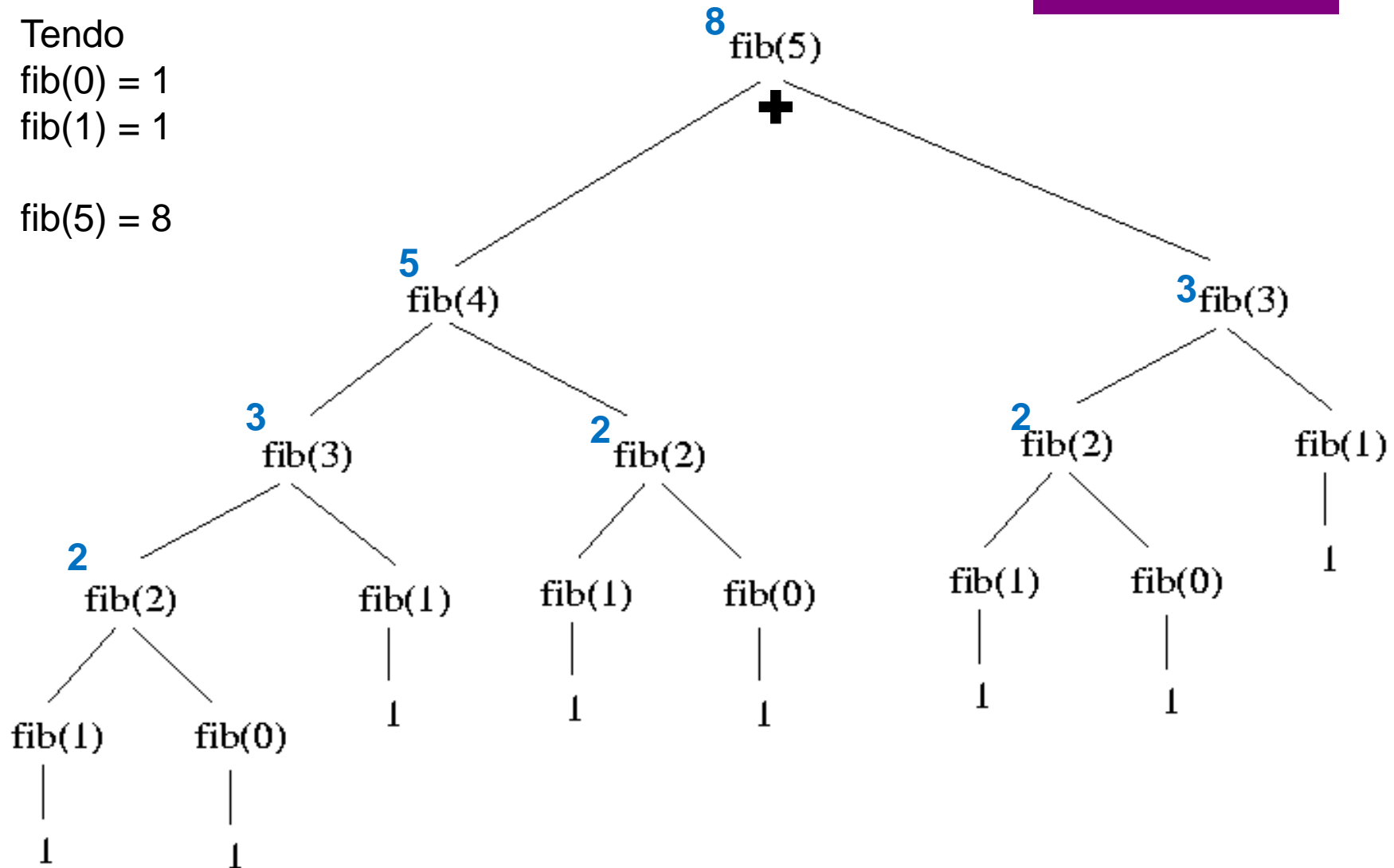
## 6. Recursividade: Exemplo Fibonacci

Tendo

$\text{fib}(0) = 1$

$\text{fib}(1) = 1$

$\text{fib}(5) = 8$



# 7. Exemplo

- 1) Escreva uma **função recursiva** para calcular o valor de uma base x elevada a um expoente y.

```
#include <stdio.h>
int potencia(int base, int exp){
    if (exp == 0)
        return 1;
    else
        return (base * potencia(base, exp-1));
}

int main(){
    int base, expoente, ret;
    scanf("%d %d", &base, &expoente);

    ret = potencia(base, expoente);

    printf("Resultado: %d", ret);
    return 0;
}
```

# 7. Exercícios

---

## 1) BEE 1029 - Fibonacci (**recursivo**)

<https://judge.beecrowd.com/pt/problems/view/1029>

# ***Complexidade de Tempo***



# 8. Número de Operações Relevantes

---

## MOTIVAÇÃO

- Será que vale a pena construir determinados programas?
- E se o programa demorar muito para terminar de rodar?
- E se houver vários programas que resolvem o mesmo problema, mas com implementações diferentes... qual escolher?

# 8. Número de Operações Relevantes

---

- Todo programa possui operações (comandos do código) que se transformarão em comandos de máquina.
- Cada operação do programa transforma-se em uma, ou mais, instruções para o processador.
- Quanto mais instruções um programa gera, mais tempo ele vai levar para terminar sua execução!

# 8. Número de Operações Relevantes

---

- Exemplo 1: Quantas operações relevantes faz o programa abaixo?

```
int main() {  
    int a = 2;  
    int b = 3;  
    int c;  
    c = (a + b)/2;  
    printf("%d\n", c);  
    return 0;  
}
```



# 8. Número de Operações Relevantes

## ■ Exemplo 1: Quantas operações relevantes faz o programa abaixo?

```
int main() {  
    int a = 2; ..... ➔ 1 operação (atribuição)  
    int b = 3; ..... ➔ 1 operação (atribuição)  
    int c; ..... ➔ 0 operações (apenas declaração - não contamos!)  
    c = (a + b) / 2; ..... ➔ 3 operações (uma soma, uma divisão e uma atribuição)  
    printf("%d\n", c); ..... ➔ 1 operação (vamos considerar apenas uma!)  
    return 0; ..... ➔ 1 operação (retorno de valor)  
}
```

Total de:  $1 + 1 + 3 + 1 + 1 = 7$  operações relevantes.

# 8. Número de Operações Relevantes

- Exemplo 2: Quantas operações relevantes faz a função abaixo?

```
int main() {  
    int a = 1;  
    int i = 0;  
    while (i < 100) {  
        a += i++;  
    }  
    printf("%d\n", a);  
    return 0;  
}
```

# 8. Número de Operações Relevantes

## ■ Exemplo 2: Quantas operações relevantes faz a função abaixo?

```
int main() {  
    int a = 1; ..... 1 operação (atribuição)  
    int i = 0; ..... 1 operação (atribuição)  
    while (i < 100) { ..... 101 operações de comparação  
        a += i++;  
    }  
    printf("%d\n", a);  
    return 0;  
}
```

(a primeira comparação é quando i vale 0; a última é quando i vale 100, terminando o while – de 0 a 100, i mudou 101 vezes).

a += i++ soma 4 operações: a = a + i e i = i + 1  
a += i++ acontece para cada valor de i (de 0 a 99)  
logo, total de  $4 \times 100 = 400$  operações.

1 operação      1 operação

Total de:  $1 + 1 + 101 + 400 + 1 + 1 = 505$  operações relevantes.

# 8. Número de Operações Relevantes

- Exemplo 3: Quantas operações relevantes faz a função abaixo?

```
int soma(int a, int n) {  
    int i;  
    int s = 0;  
    for (i = 0; i < n; i++) {  
        s = s + a;  
    }  
    return s;  
}
```

# 8. Número de Operações Relevantes

## ■ Exemplo 3: Quantas operações relevantes faz a função abaixo?

```
int soma(int a, int n) {  
    int i; -----> 0 operações (apenas declaração)  
    int s = 0; -----> 1 operação (atribuição)  
    for (i = 0; i < n; i++) { -----> 1 operação (executa apenas uma vez i = 0) +  
        s = s + a; -----> n+1 comparações (executa i < n n+1 vezes) +  
    } -----> n incrementos (executa n vezes i++)  
    return s;  
}
```

(mas cada i++ representa duas operações: soma e atribuição: i=i+1; logo, i++ soma 2n operações).

total para o for: **3n + 2** operações

2 operações para cada valor de i  
i varia de 0 até n-1 (muda n vezes)  
portanto, isto dá um total de **2n** operações.

1 operação

Total de:  $1 + (3n + 2) + 2n + 1 = 5n + 4$  operações relevantes.

# 8. Número de Operações Relevantes

■ **Exercício 2:** Quantas operações relevantes faz a função abaixo?

```
int soma(int a, int n) {  
    int i, j, s = 0;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n, j++)  
            s += a;  
    return s;  
}
```

# 9. Tempo de Execução

---

- Se soubermos quantas operações / instruções um programa gera e, se também soubermos quanto tempo um processador leva para processar uma dessas operações / instruções, então, podemos calcular o tempo que o programa vai demorar para terminar.
- Os cálculos, normalmente, envolvem conceitos matemáticos como regras de três simples e potências de 10 (notação científica).

# 9. Tempo de Execução

## ■ Lembrete: Potências de 10

Potência	Símbolo	Nome	Valor
$10^{15}$	<b>P</b>	Peta	1.000.000.000.000.000
$10^{12}$	<b>T</b>	Tera	1.000.000.000.000
$10^9$	<b>G</b>	Giga	1.000.000.000
$10^6$	<b>M</b>	Mega	1.000.000
$10^3$	<b>K</b>	Kilo	1.000
$10^0$	-	-	1
$10^{-3}$	<b>m</b>	mili	0,001
$10^{-6}$	<b><math>\mu</math></b>	micro	0,000001
$10^{-9}$	<b><math>\eta</math></b>	nano	0,000000001
$10^{-12}$	<b><math>\rho</math></b>	pico	0,0000000000001
$10^{-15}$	<b>f</b>	femto	0,0000000000000001



# 9. Tempo de Execução

- Mas, como calcular o tempo de execução de um programa?
- Se tomarmos, por exemplo, o Exemplo 1: nele, o programa faz 7 operações relevantes. Logo, se ele fosse executado em um máquina com processador, por exemplo, de 100MHz, quanto tempo o programa iria demorar para terminar sua execução?

```
int main() {  
    int a = 2; ..... ➔ 1 operação (atribuição)  
    int b = 3; ..... ➔ 1 operação (atribuição)  
    int c; ..... ➔ 0 operações (apenas declaração - não contamos!)  
    c = (a + b) / 2; ..... ➔ 3 operações (uma soma, uma divisão e uma atribuição)  
    printf("%d\n", c); ..... ➔ 1 operação (vamos considerar apenas uma!)  
    return 0; ..... ➔ 1 operação (retorno de valor)  
}
```

# 9. Tempo de Execução

- Dizer que o processador tem velocidade de 100MHz, significa dizer que o processador realiza 100 *Mega* ciclos de clock por segundo ( $100 \times 10^6$  ciclos/seg).
- Podemos considerar que um processador processa uma operação / instrução relevante a cada ciclo de clock <sup>1</sup>. Logo, o processador de 100MHz irá processar 100 *Mega* operações / instruções relevantes por segundo.
- Logo, se o nosso programa faz 7 operações relevantes, como calcular quanto tempo ele vai demorar para terminar se ele for executado na máquina com processador de 100MHz?

---

<sup>1</sup> Na verdade, a velocidades dos processadores hoje em dia é medida em **MIPS** (Milhões de Instruções Por Segundo), que, por causa de várias otimizações do processador, torna-se, em geral, um valor mais alto do que o valor da frequência (MHz). Veja em: [https://pt.wikipedia.org/wiki/MIPS\\_\(medida\)](https://pt.wikipedia.org/wiki/MIPS_(medida))

# 9. Tempo de Execução

- 7 operações relevantes em um processador de 100 MHz

operações                      tempo  
 $100 \times 10^6$   $\longrightarrow$  1 segundo  
7                       $\longrightarrow$  **X**

$$\begin{aligned} 100 \times 10^6 \text{ X} &= 7 \times 1 \\ 10^8 \text{ X} &= 7 \\ \text{X} &= 7 / 10^8 \\ \text{X} &= \mathbf{7 \times 10^{-8} \text{ seg.}} \end{aligned}$$

► 70 nanosegundos

$10^0$	-	-	1
$10^{-3}$	<b>m</b>	<u>mili</u>	0,001
$10^{-6}$	<b>μ</b>	micro	0,000001
$10^{-9}$	<b>η</b>	<u>nano</u>	0,000000001

# 9. Tempo de Execução

---

- **Exercício 3:** Considere que a implementação de um certo algoritmo de pesquisa em vetores realiza  $2n + 10$  operações relevantes. Dado um vetor de tamanho  $n = 495$  e sabendo-se que este programa de pesquisa será executado em um processador de 200MHz, calcule o tempo de execução do programa.

# 10. Notação Big-Oh

---

- Pensando logicamente, quanto mais operações relevantes um algoritmo faz, mas tempo ele gasta para rodar.
- Mas, será que um algoritmo que realiza  $(5n + 4)$  operações é tão melhor assim do que um que realiza  $(8n + 10)$ ?
- E um algoritmo que realiza  $(n^2 + n)$  é tão pior assim que um algoritmo que realiza  $(n^2 + 14)$ ?
- Resposta: depende do valor de  $n$ .

# 10. Notação Big-Oh

- Quantidade de operações realizadas pelos algoritmos:

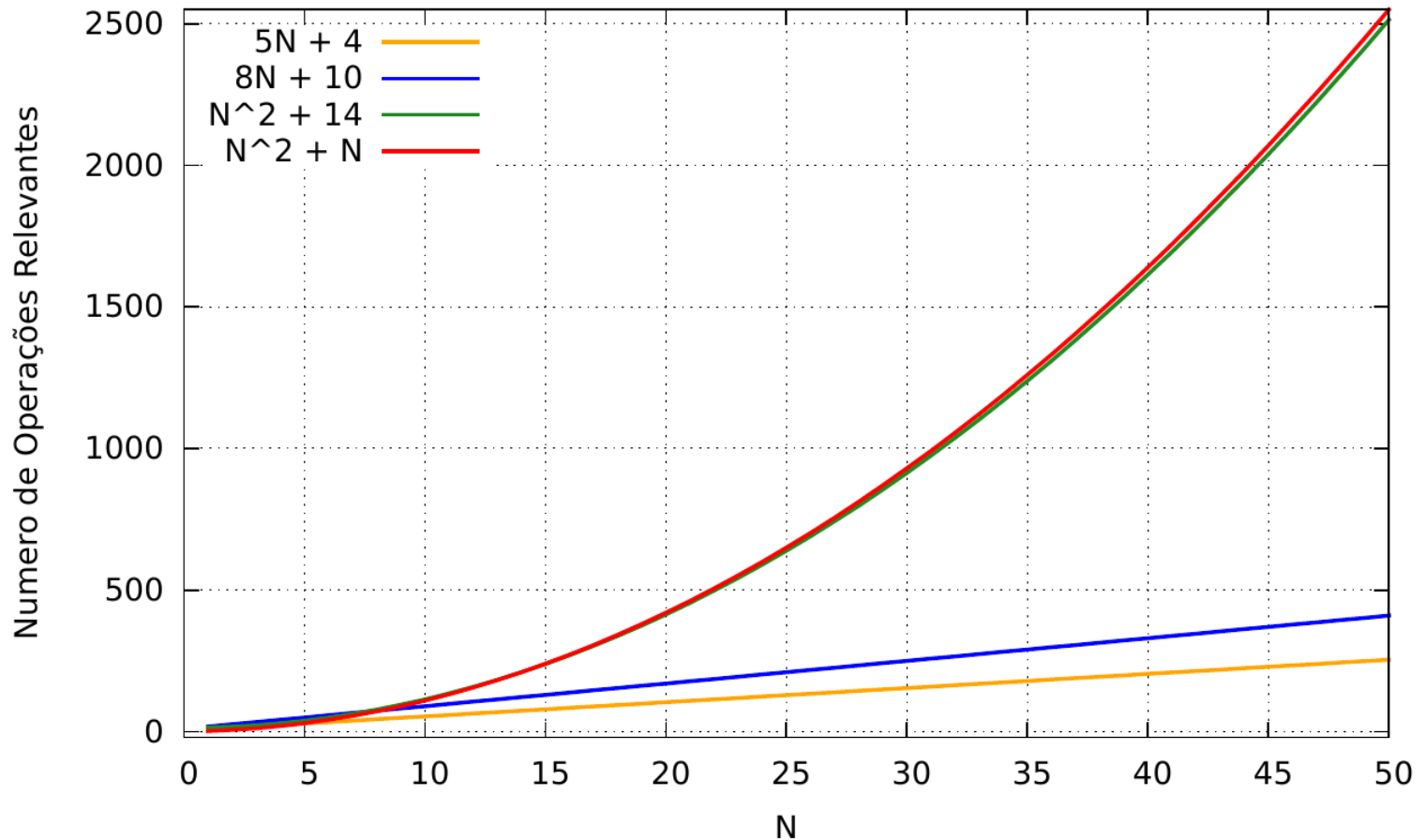
valor de $n$	$5n + 4$	$8n + 10$	$n^2 + 14$	$n^2 + n$
5	29	50	39	30
10	54	90	114	110
15	79	130	239	240
20	104	170	414	420
25	129	210	639	650
30	154	250	914	930
35	179	290	1239	1260
40	204	330	1614	1640
45	229	370	2039	2070
50	254	410	2514	2550

# 10. Notação Big-Oh

- Na tabela anterior, quando  $n$  tem valor baixo, o algoritmo que gasta mais tempo é o que realiza  $(8n + 10)$  operações. Mas, ao passo em que  $n$  cresce, fica evidente que o algoritmo que gasta mais tempo é o que realiza  $(n^2 + n)$  operações.
- Nota-se, também, que, enquanto  $n$  cresce, a diferença entre os algoritmos  $(5n + 4)$  e  $(8n + 10)$  não fica tão grande. Da mesma forma, a diferença entre  $(n^2 + 14)$  e  $(n^2 + n)$  também não fica tão grande.
- Mas, a diferença entre os algoritmos que possuem o termo  $n^2$  e os que possuem apenas o termo  $n$  fica cada vez maior, ao passo em que  $n$  cresce. De fato, podemos observar isto em um gráfico!

# 10. Notação Big-Oh

- Variação do tempo gasto pelos algoritmos de acordo com o tamanho de  $N$





# 10. Notação Big-Oh

- A **Complexidade de Tempo** de um algoritmo é indicada pelo fator (termo) que mais influencia no gasto de tempo, quando o tamanho da entrada ( $n$ ) cresce.
  - Para os algoritmos ( $5n + 4$ ) e ( $8n + 10$ ), o fator que mais influencia no gasto de tempo é  $n$ .
  - Já, para os algoritmos ( $n^2 + 14$ ) e ( $n^2 + n$ ), o fator que mais influencia no gasto de tempo é  $n^2$ .
- Note, portanto, que os números (constantes) são desconsiderados, pois eles não influenciam ou influenciam muito pouco no gasto de tempo. E, no caso de uma soma de termos não constantes ( $n^2 + n$ ), leva-se em conta apenas o termo mais relevante ( $n^2$ ).

# 10. Notação Big-Oh

- O fator (termo) que mais influencia no gasto de tempo de um algoritmo é chamado de “**ordem de tempo**” (ou “ordem de complexidade”) do algoritmo e indica a taxa de crescimento do tempo quando o tamanho da entrada (quantidade de dados) cresce.
  - $(5n + 4)$ : algoritmo “da ordem de”  $n$ .
  - $(8n + 10)$ : algoritmo “da ordem de”  $n$ .
  - $(n^2 + 14)$ : algoritmo “da ordem de”  $n^2$ .
  - $(n^2 + n)$ : algoritmo “da ordem de”  $n^2$ .

# 10. Notação Big-Oh

---

- E o algoritmo que fez **7 operações relevantes**? Qual é a sua ordem de tempo / complexidade?
- Resposta: quando o tempo gasto de um algoritmo não depende do tamanho da entrada (é sempre um número constante), dizemos que o algoritmo é “**da ordem de**” 1.

# 10. Notação Big-Oh

- A notação  $O()$  (Big-Oh) é utilizada para indicar a “curva” (ou seja, a tendência) da complexidade computacional de um algoritmo considerando o **pior caso** possível (ou seja, a situação em que o algoritmo gasta o máximo de tempo), conforme o tamanho da entrada cresce (conforme aumenta o valor de  $n$ ).

7:	“da ordem de” 1	=	$\mathcal{O}(1)$	(complexidade <i>constante</i> )
$5n + 4$ :	“da ordem de” $n$	=	$\mathcal{O}(n)$	(complexidade <i>linear</i> )
$8n + 10$ :	“da ordem de” $n$	=	$\mathcal{O}(n)$	(complexidade <i>linear</i> )
$n^2 + 14$ :	“da ordem de” $n^2$	=	$\mathcal{O}(n^2)$	(complexidade <i>quadrática</i> )
$n^2 + n$ :	“da ordem de” $n^2$	=	$\mathcal{O}(n^2)$	(complexidade <i>quadrática</i> )

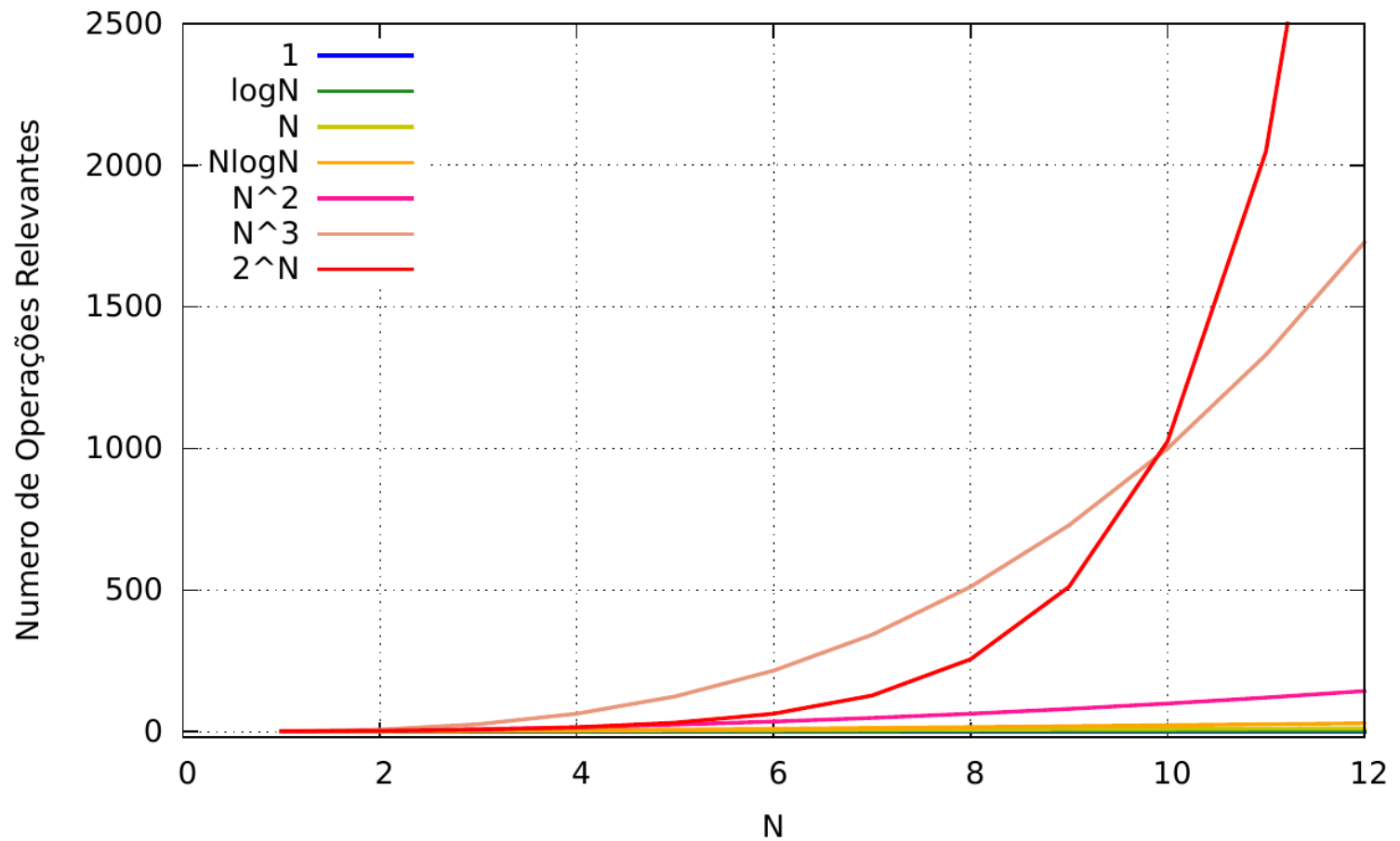
# 10. Notação Big-Oh

- Principais ordens de tempo (ordenadas da “melhor” para a “pior”)

$\mathcal{O}(1)$ :	complexidade <i>constante</i>
$\mathcal{O}(\log N)$ :	complexidade <i>logarítmica</i>
$\mathcal{O}(N)$ :	complexidade <i>linear</i>
$\mathcal{O}(N \log N)$ :	complexidade <i><math>n \log n</math></i>
$\mathcal{O}(N^2)$ :	complexidade <i>quadrática</i>
$\mathcal{O}(N^3)$ :	complexidade <i>cúbica</i>
$\mathcal{O}(2^N)$ :	complexidade <i>exponencial</i>
$\mathcal{O}(N!)$ :	complexidade <i>fatorial</i>
$\mathcal{O}(N^N)$ :	complexidade <i>exponencial</i> (pior caso)
$\mathcal{O}(N^{N^N})$ :	complexidade <i>duplamente exponencial</i>

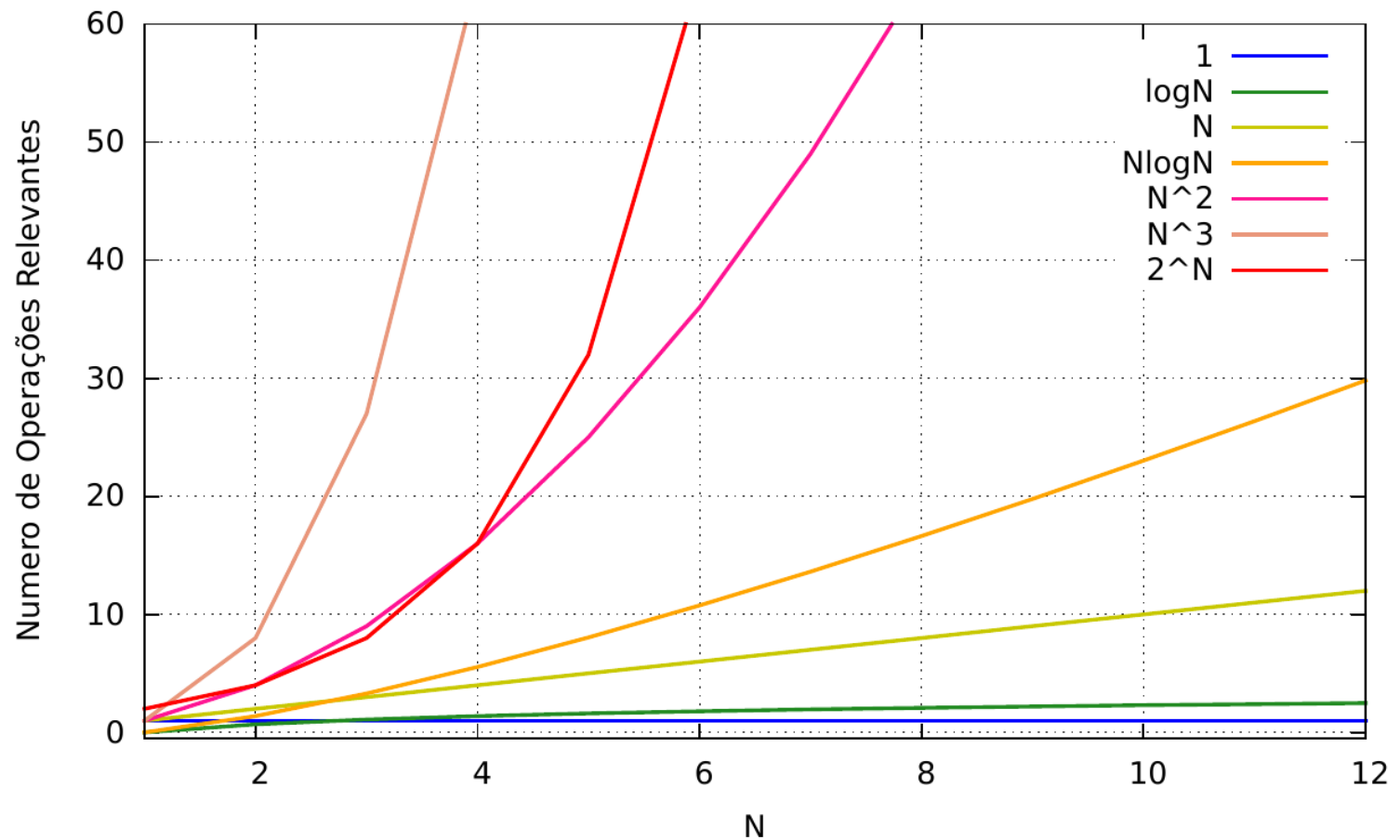
# 10. Notação Big-Oh

- Variação das ordens de tempo de acordo com o tamanho de  $N$  da entrada



# 10. Notação Big-Oh

- Variação das ordens de tempo (zoom do gráfico anterior)

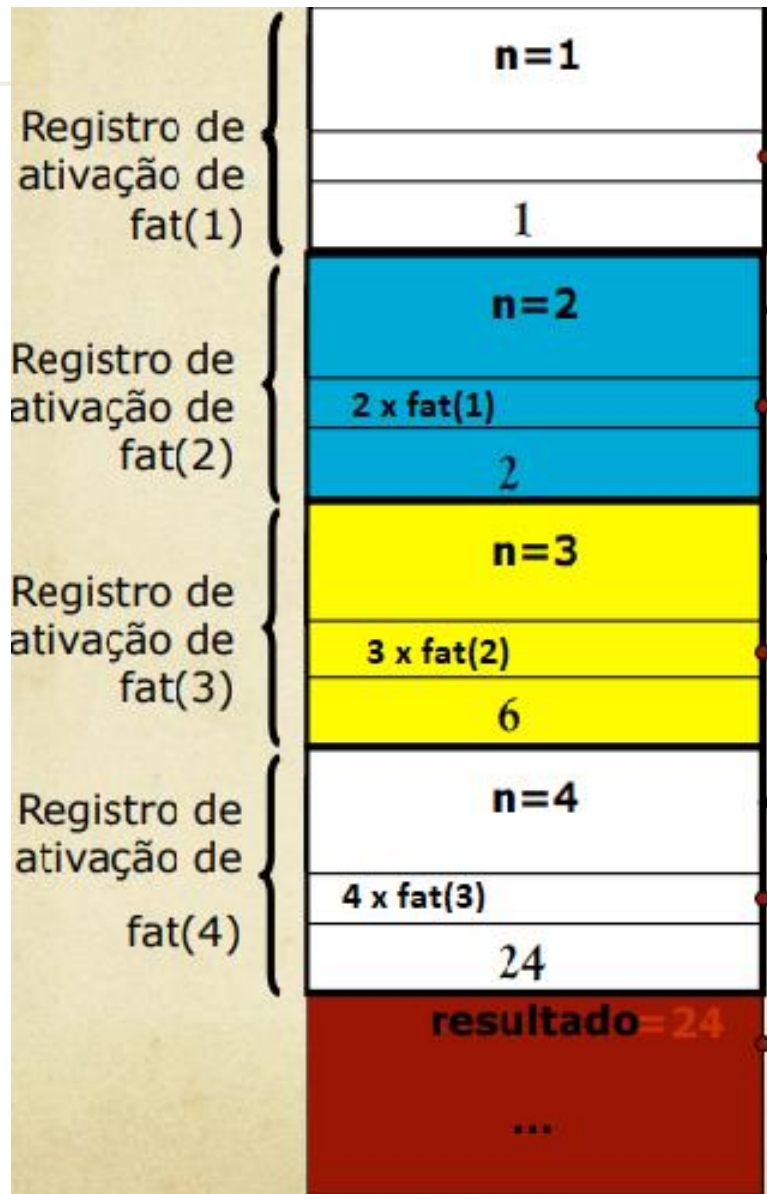




# ***Terceiro momento***



# Terceiro momento: síntese



## ■ Recursividade

$$4! = 4 * 3 * 2 * 1$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1$$

# Terceiro momento: síntese

---

- Todo programa possui operações (comandos do código) que transformam-se em uma, ou mais, instruções para o processador.
- A partir da quantidade de operações e do tempo o processador leva para processar uma dessas operações, então é possível calcular o tempo que o programa vai demorar para terminar.
- A notação  $O()$  (Big-Oh) é utilizada para indicar a “curva” da complexidade computacional de um algoritmo considerando o **pior caso** possível conforme o tamanho da entrada cresce (conforme aumenta o valor de  $n$ ).