

Aula anterior

- Recursividade

AEDs II – Noções de Complexidade de Algoritmos

Prof. Diego Silva Caldeira Rocha

Objetivos

Noções de Complexidade

- Avaliação de Desempenho
- Notação Big O
- Notação Theta Θ
- Notação Omega Ω
- Análise de algoritmos

Avaliação de Desempenho

A avaliação de desempenho de um algoritmo quando executado por um computador pode ser realizada:

- I. **a posteriori**
- II. **a priori**

Avaliação de Desempenho

II – Avaliação de desempenho a posteriori:

- envolve a execução propriamente dita do algoritmo, medindo-se seu tempo de execução
- Só pode ser exata se forem conhecidos os detalhes da arquitetura da máquina, da linguagem de programação utilizada, do código gerado pelo compilador, etc.
- Dessa forma, o tempo de execução será diferente para cada algoritmo, compilador e computador

Avaliação de Desempenho

I – Avaliação de desempenho a priori:

- É realizada de forma analítica **sem a execução do algoritmo**
- Podem ser considerarmos dois itens:
 - ✓ a entrada (os dados fornecidos)
 - ✓ e o número de instruções executadas pelo algoritmo
- Em geral, o aspecto importante da entrada é seu “**tamanho**”, que pode ser dado como:
 - ✓ número de valores contidos em um vetor
 - ✓ o número de registros contidos em um arquivo
 - ✓ ou seja, um certo número de elementos que constituem a entrada de dados para o algoritmo

Como medir o custo de execução de um algoritmo?

Função de Custo ou Função de Complexidade

- $T(n)$ = medida de custo necessário para executar um algoritmo para um problema de tamanho n
- Se $T(n)$ é uma medida da quantidade de tempo necessário para executar um algoritmo para um problema de tamanho n , então T é chamada ***função de complexidade de tempo de algoritmo***
- Se $T(n)$ é uma medida da quantidade de memória necessária para executar um algoritmo para um problema de tamanho n , então T é chamada ***função de complexidade de espaço de algoritmo***

Como medir o custo de execução de um algoritmo?

Observação: tempo não é tempo!

- É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, **mas o número de vezes que determinada operação considerada relevante é executada**

Custo Assintótico de Funções

Na análise de algoritmos, o interessante é comparar algoritmos para *valores grandes de n* podendo ignorar valores pequenos de n

Essa análise matemática, voltada somente para valores muito grandes de n , é denominada: **Análise Assintótica**

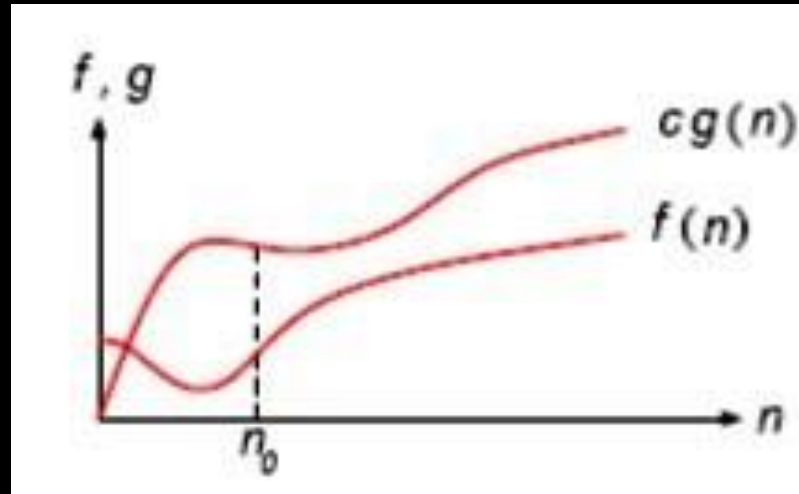
Notação assintótica de funções

Existem três notações principais na análise de assintótica de funções:

- Notação O (“ O ” grande) ou Big O
- Notação Θ
- Notação Ω

Notação O grande

- A notação **O** define um **limite superior** para a função, por um valor constante
- Escreve-se **$f(n) = O(g(n))$** , se existirem constantes positivas **c** e **n_0** tais que para **$n \geq n_0$** , o valor de **$f(n)$** é menor ou igual a **$cg(n)$**



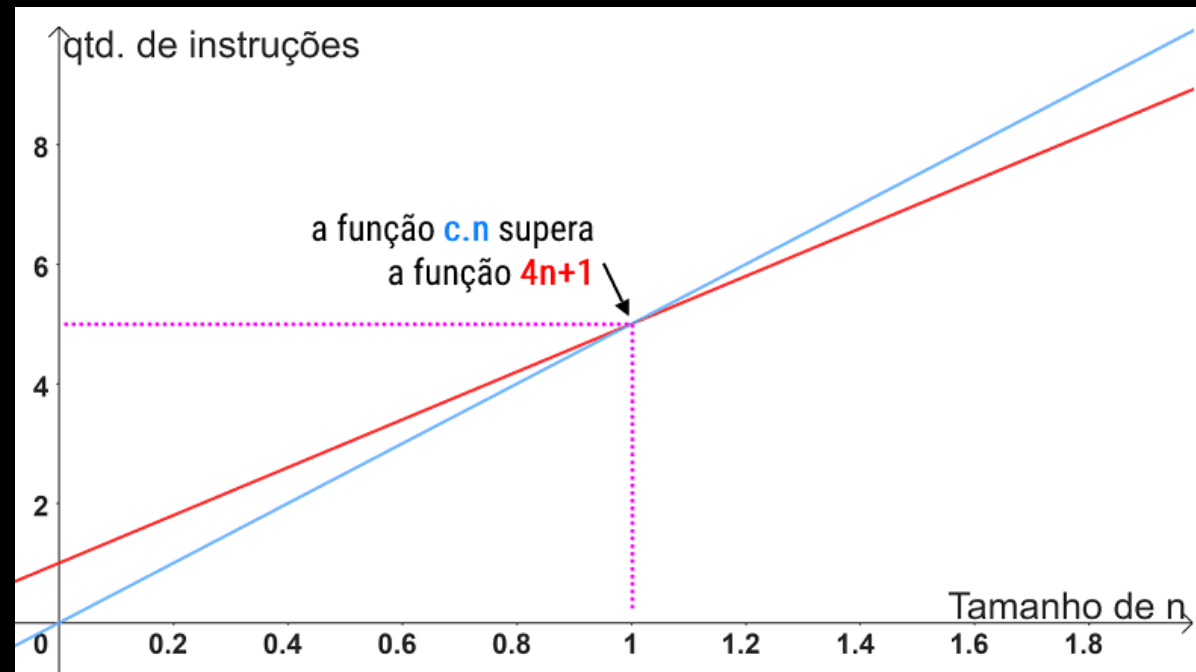
O grande Demonstração (I)

- Demonstrar pela definição que $4n+1=O(n)$
- $f(n) = 4n+1$
- $g(n) = n$
- pela definição matemática, precisamos demonstrar que existe uma constante positiva c e um valor positivo n_0 inicial, de forma que:
- $f(n) \leq c.g(n)$ para todos os $n \geq n_0$.
- LOGO $4n+1 \leq c.n$ para todos os $n \geq n_0$

O grande Demonstração (I)

- Para tanto, a primeira decisão que precisamos tomar é escolher um valor para a constante c . Neste caso, vamos considerar c igual ao valor 5. Observe abaixo uma tabela que mostra os resultados da desigualdade para $c=5$ e para alguns valores de n .

n	$4n + 1$		$c \cdot n$
0	1	$>$	0
1	5	$=$	5
2	9	$<$	10
3	13	$<$	15
4	17	$<$	20
5	21	$<$	25



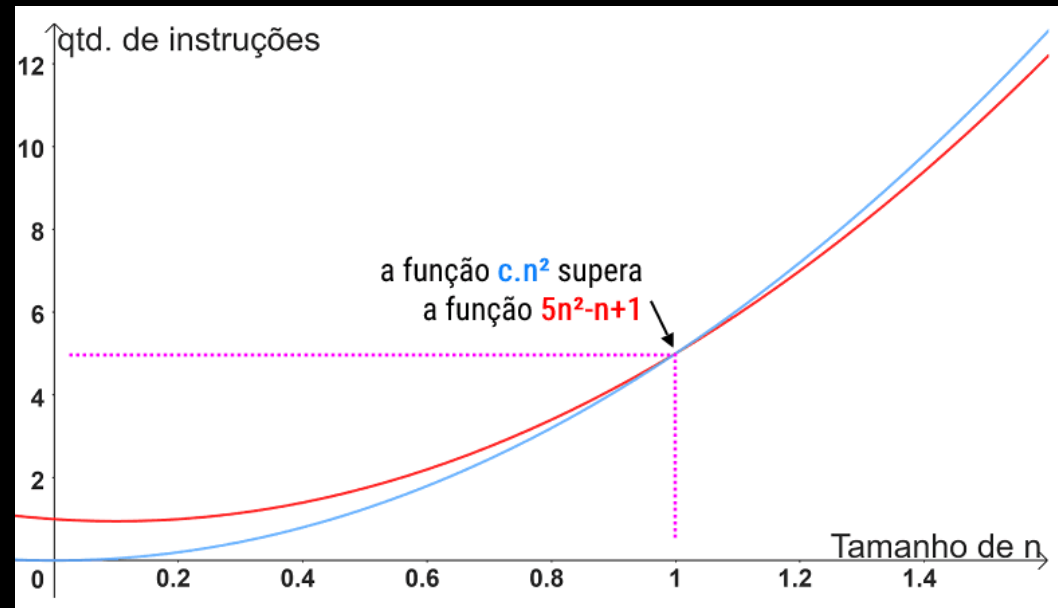
O grande Demonstração (II)

- Demonstrar pela definição que $5n^2 - n + 1 = O(n^2)$
- $f(n) = 5n^2 - n + 1$
- $g(n) = n^2$
- pela definição matemática, precisamos demonstrar que existe uma constante positiva c e um valor positivo n_0 inicial, de forma que:
- $f(n) \leq c \cdot g(n)$ para todos os $n \geq n_0$.
- LOGO $5n^2 - n + 1 \leq cn^2$ para todos os $n \geq n_0$

O grande Demonstração (II)

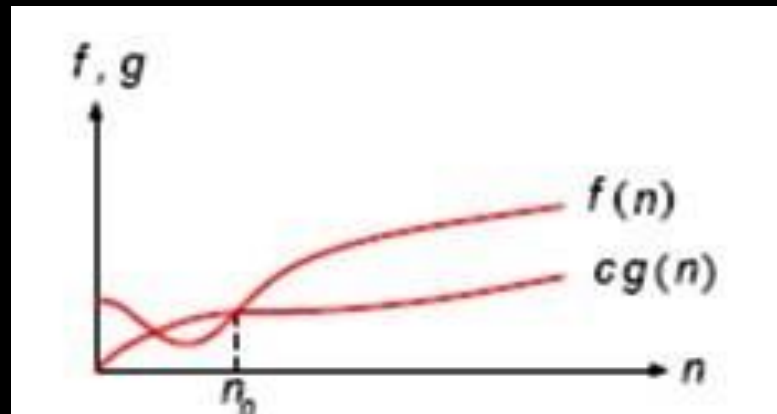
- Para tanto, a primeira decisão que precisamos tomar é escolher um valor para a constante c . Neste caso, vamos considerar c igual ao valor 5. Observe abaixo uma tabela que mostra os resultados da desigualdade para $c=5$ e para alguns valores de n .

n	$5n^2 - n + 1$		$c \cdot n^2$
0	1	$>$	0
1	5	$=$	5
2	19	$<$	20
3	43	$<$	45
4	77	$<$	80
5	121	$<$	125



Notação Ω

- A notação Ω define um **limite inferior** para a função, por um valor constante
- Escreve-se $f(n) = \Omega(g(n))$, se existirem constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é maior ou igual a $cg(n)$

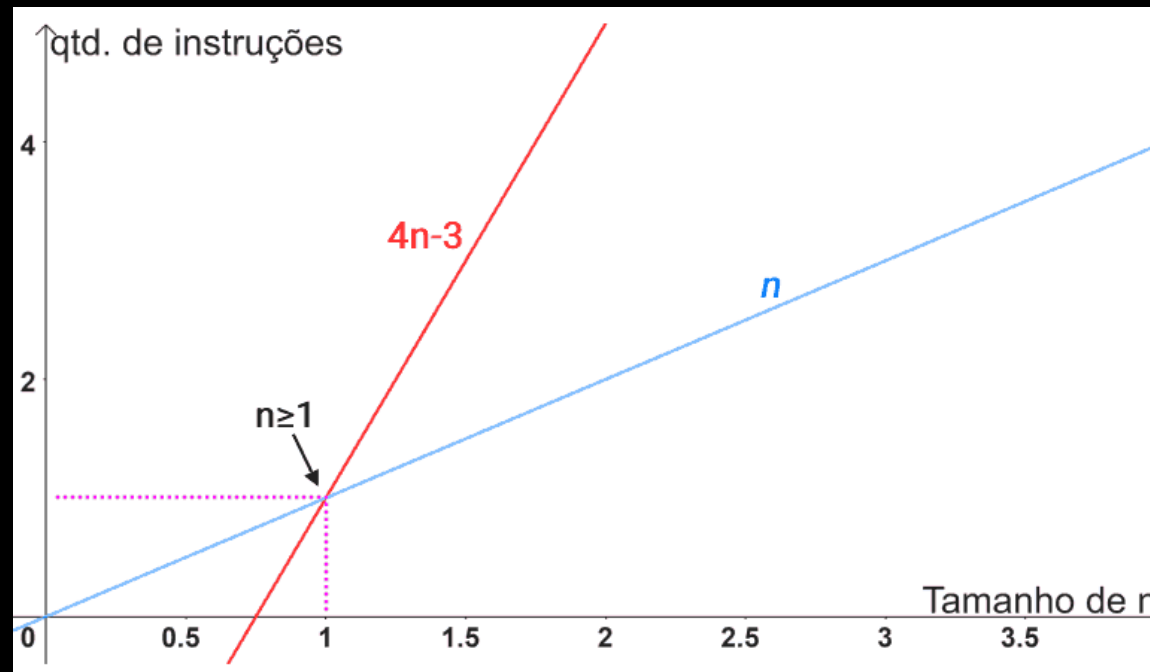


Notação Ω Demonstração (I)

- Demonstrar pela definição que $4n-3 = \Omega(n)$
- $f(n) = 4n-3$
- $g(n) = n$
- $\Omega(n)$ apresenta-se como um limite inferior, e sabemos que a função $4n-3$ nunca apresentará um comportamento de crescimento que seja ultrapassado por esse limite inferior.
- $c \cdot g(n) \leq f(n)$ para todos os $n \geq n_0$.
- LOGO $c \cdot n \leq 4n-3$ para todos os $n \geq n_0$

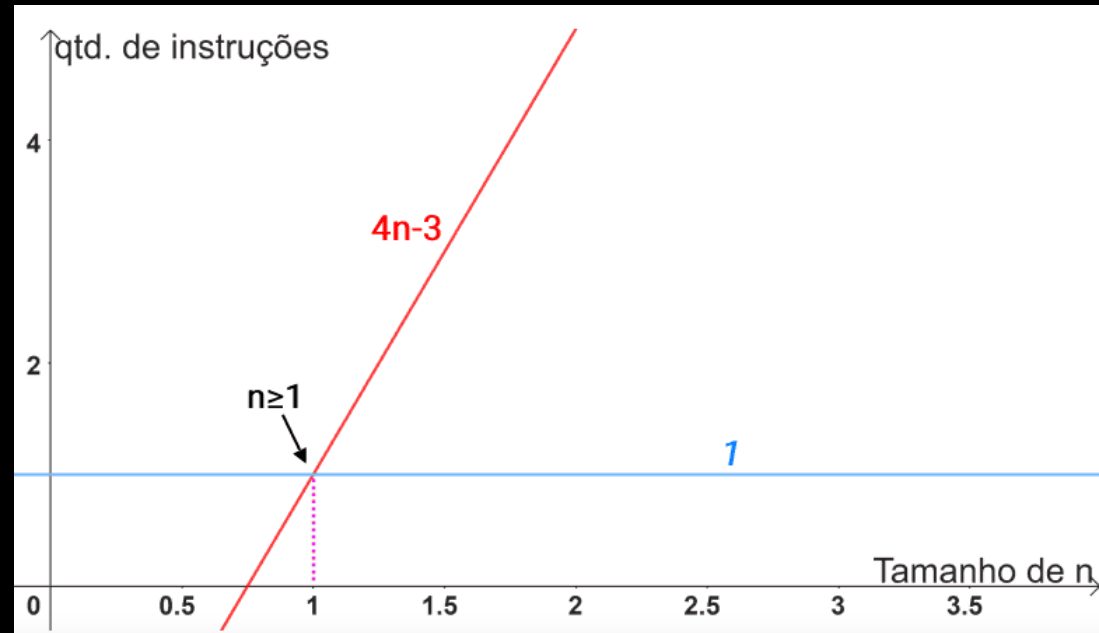
Notação Ω Demonstração (I)

- Exemplo: para a função $4n-3$, existe uma outra função de comportamento linear que a limita inferiormente. Perceba no gráfico abaixo que, para valores de $n \geq 1$, a função $4n-3$ supera a função n .



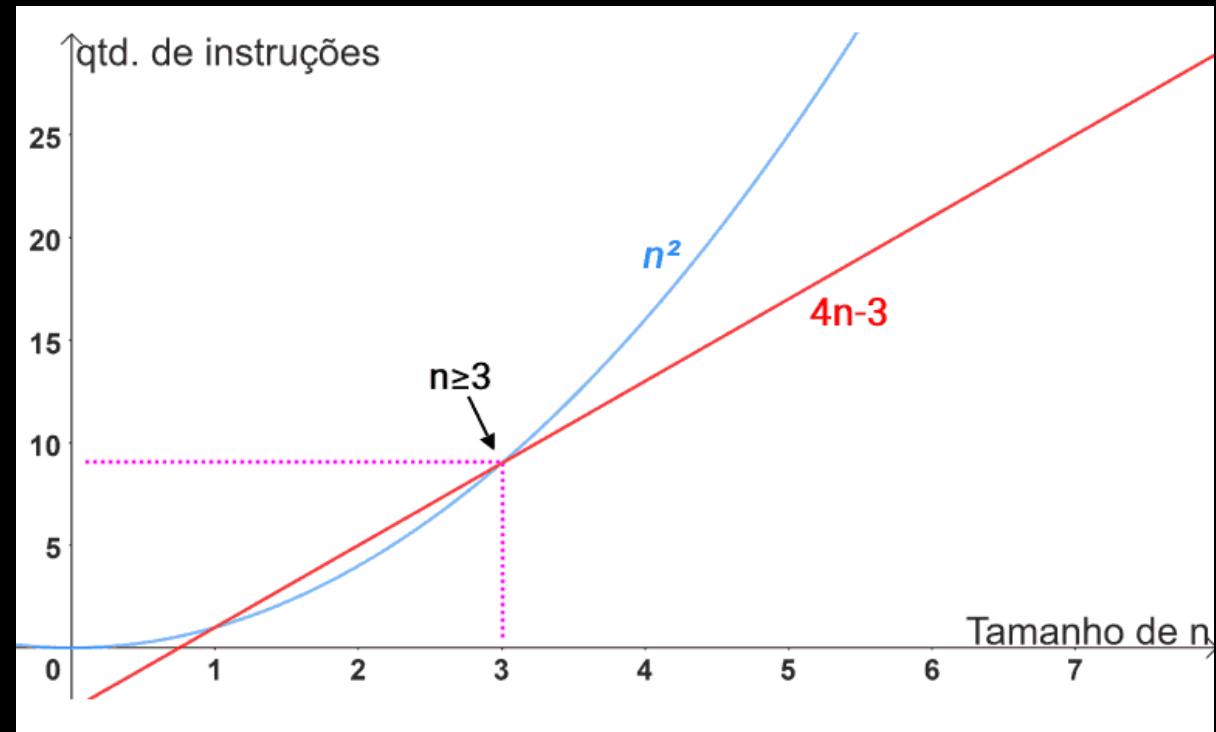
Notação Ω Demonstração (I)

- Da mesma forma, também é correto afirmar que $4n-3$ é $\Omega(1)$, pois a função $4n-3$ nunca apresentará um comportamento de crescimento que seja ultrapassado por um comportamento constante. Desta forma, $\Omega(1)$ também se apresenta como um limite assintótico inferior:



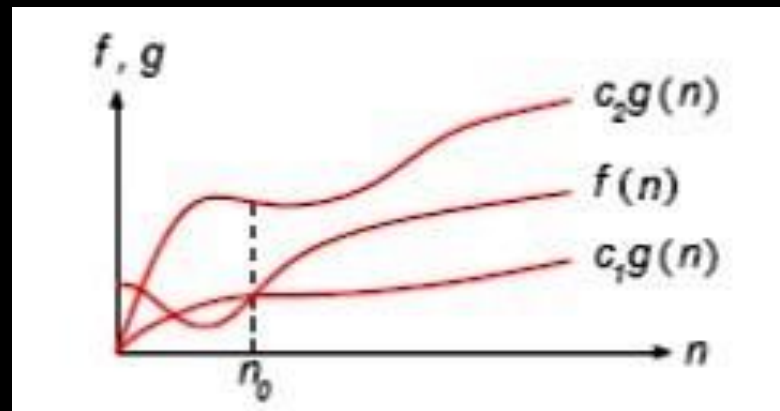
Notação Ω Demonstração (I)

- Agora, seria errado dizer que $4n-3$ é $\Omega(n^2)$, pois a função $4n-3$ nunca crescerá a ponto de ultrapassar o comportamento quadrático. Desta forma, $\Omega(n^2)$ não se apresenta como um limite assintótico inferior:



Notação Θ

- A notação Θ limita a função por fatores constantes
- Escreve-se $f(n) = \Theta(g(n))$, se existirem constantes positivas c_1 , c_2 e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ está sempre entre $c_1g(n)$ e $c_2g(n)$ inclusive



Notação Θ Demonstração (I)

- Demonstrar pela definição que $4n+1=\Theta(n)$
- $f(n) = 4n+1$
- $g(n) = n$
- Precisamos demonstrar que existem duas constantes positivas c_1 e c_2 e um valor n_0 inicial, de forma que:
- $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ para todos os $n \geq n_0$.
- LOGO $c_1 \cdot n \leq 4n+1 \leq c_2 \cdot n$ para todos os $n \geq n_0$

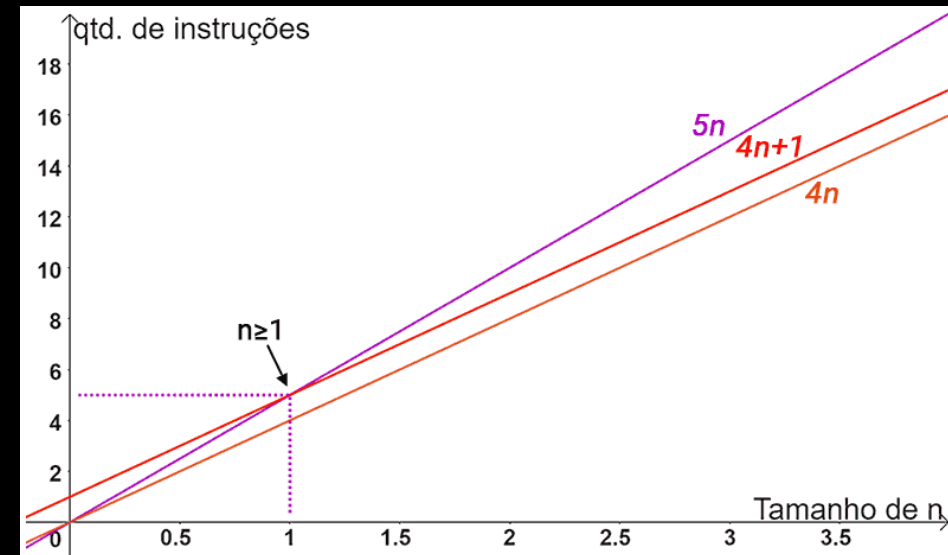
Notação Θ Demonstração (I)

- Em resumo: precisamos provar que, para todos os valores de n maiores ou igual a n_0 , a função $4n+1$ é:
- maior ou igual a c_1 multiplicado por n e;
- menor ou igual a c_2 multiplicado por n .

Notação Θ Demonstração (I)

- Para tanto, a primeira decisão que precisamos tomar é escolher valores para as constantes c_1 e c_2 . Neste caso, vamos considerar c_1 sendo igual ao valor 4 e c_2 sendo igual ao valor 5. Observe abaixo uma tabela que mostra os resultados da desigualdade para alguns valores de n :

n	$4n$		$4n + 1$		$5n$
1	4	<	5	=	5
2	8	<	9	<	10
3	12	<	13	<	15
4	16	<	17	<	20
5	20	<	21	<	25



Notação Θ Demonstração (I)

- Isso significa que a função $4n+1$ nunca terá um comportamento de crescimento inferior a $4n$ e nem crescerá mais que $5n$. Por este motivo, dizemos que $\Theta(n)$ representa um limite assintótico justo para a função $4n+1$, porque ela é limitada superiormente e inferiormente por duas outras funções de uma mesma classe assintótica que ela: a classe linear n .
- Desta forma, conseguimos provar o nosso objetivo. Existem duas constantes positivas $C_1=4$ e $C_2=5$ e um $n_0=1$, tal que $C_1 \cdot n \leq 4n+1 \leq C_2 \cdot n$ para todos os valores de n maiores ou igual a n_0 .
- Portanto, está provado que, de fato, $4n+1 = \Theta(n)$.

Análise de Algoritmos

Casos a serem analisados em uma análise:

- **Melhor caso**
- **Pior caso**
- **Caso médio**

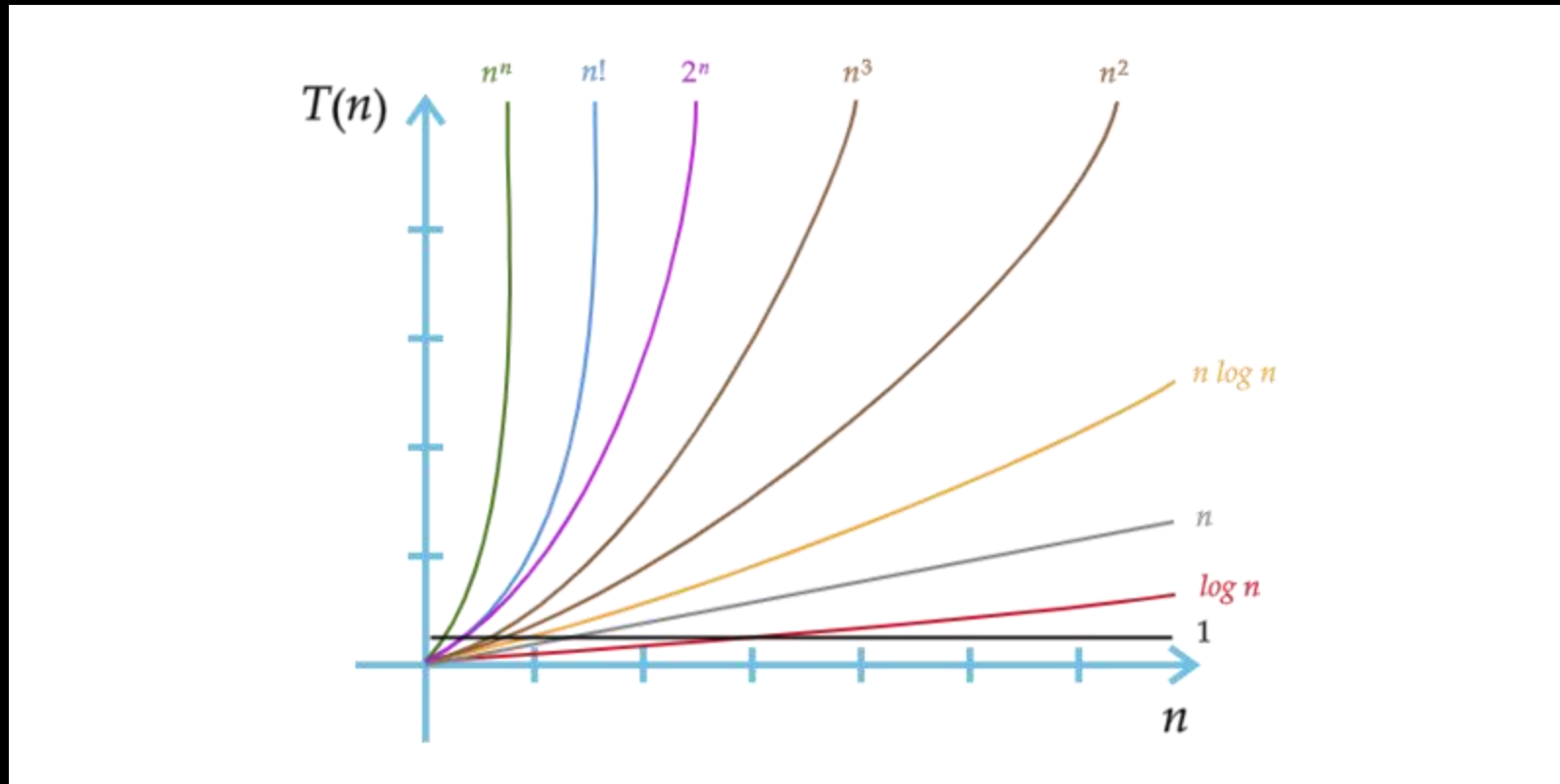
Classificação das Funções

- **$O(1)$ – complexidade constante:**
 - O tempo de execução independe do número de elementos na entrada
- **$O(\log n)$ – complexidade logarítmica:**
 - O algoritmo resolve um problema transformando-o em partes menores
- **$O(n)$ – complexidade linear:**
 - Um pequeno trabalho é realizado sobre cada elemento da entrada
- **$O(n \log n)$**
 - O algoritmo quebra um problema em partes menores, resolve cada uma separadamente e depois junta as soluções

Classificação das Funções

- **$O(n^2)$ – complexidade quadrática:**
 - Elementos processados aos pares
- **$O(n^3)$ – complexidade cúbica:**
 - Tipicamente, multiplicações de matrizes
 - Úteis apenas para problemas pequenos
- **$O(2^n)$ – complexidade exponencial**
- **$O(n!)$ – complexidade fatorial:**
 - Algoritmos de força bruta: tentam todas as possibilidades para problemas de otimização combinatória

Classificação da funções (Diferença gráficas)



Notação O – Regras Práticas

a) Regra da complexidade polinomial

Se $f(n)$ é um polinômio de grau k , então $f(n) = O(n^k)$

b) Regra da constante

$O(c * f(n)) = c * O(f(n)) = O(f(n))$

c) Regra da soma de tempos

se $T_1(n) = O(f(n))$ e $T_2(n) = O(g(n))$

então $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$

Isto significa que a complexidade de um algoritmo com dois trechos em sequência (com tempos de execução diferentes) é definida pelo trecho de maior complexidade

Notação O – Regras Práticas

d) Regra do produto de tempos

se $T_1(n) = O(f(n))$ e $T_2(n) = O(g(n))$

Então $T_1(n) * T_2(n) = O(f(n) * g(n))$

Isto significa que a complexidade de um algoritmo com dois trechos aninhados, em que o segundo é repetidamente executado pelo primeiro, é dada pelo produto da complexidade do trecho mais interno pela complexidade do trecho mais externo

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int n) {  
    int max = A[0];  
    int i = 1;  
    while (i <= n-1) {  
        if (A[i] > max)  
            max = A[i];  
        i = i + 1;  
    }  
    return max;  
}
```


Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

Pior caso:
Ocorre quando
o maior elemento é o
último elemento do vetor

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int n) {  
    int max = A[0];  
    int i = 1;  
    while (i <= n-1) {  
        if (A[i] > max)  
            max = A[i];  
        i = i + 1;  
    }  
    return max;  
}
```

**Contando
Operações**

Análise de Complexidade


Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

Análise de Complexidade

Exemplo


```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```



Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int n) {  
    int max = A[0];  
    int i = 1;  
    while (i <= n-1) {  
        if (A[i] > max)  
            max = A[i];  
        i = i + 1;  
    }  
    return max;  
}
```



2 operações

1 operação

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

← 2 operações

← 1 operação

← n operações

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

2 operações

1 operação

n operações

2 operações

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

Diagram illustrating the complexity analysis of the `findMax` function:

- `int max = A[0];`: 2 operações (Red arrow)
- `int i = 1;`: 1 operação (Red arrow)
- `while (i <= n-1) {`: n operações (Red arrow)
- `if (A[i] > max)`: 2 operações (Green arrow)
- `max = A[i];`: 2 operações (Green arrow)
- `i = i + 1;`: (No arrow)
- `}`: (No arrow)
- `return max;`: (No arrow)
- `}`: (No arrow)

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

<code>int max = A[0];</code>	2 operações
<code>int i = 1;</code>	1 operação
<code>while (i <= n-1) {</code>	n operações
<code> if (A[i] > max)</code>	2 operações
<code> max = A[i];</code>	2 operações
<code> i = i + 1;</code>	2 operações

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

← 2 operações

← 1 operação

← n operações

← 2 operações

← 2 operações

← 2 operações

n - 1
vezes

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int  
n) {
```

```
    int max = A[0];
```

← 2 operações

```
    int i = 1;
```

← 1 operação

```
    while (i <= n-1) {
```

← n operações

```
        if (A[i] > max)
```

← 2 operações

```
            max = A[i];
```

← 2 operações

```
            i = i + 1;
```

← 2 operações

n - 1
vezes

```
    }
```

```
    return max;
```

← 1 operação

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

← 2 operações

← 1 operação

← n operações

← 2 operações

← 2 operações

← 2 operações

n - 1
vezes

← 1 operação

Pior caso: $2 + 1 + n + 6 \cdot (n - 1) + 1$

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int  
n) {
```

```
    int max = A[0];
```

```
    int i = 1;
```

```
    while (i <= n-1) {
```

```
        if (A[i] > max)
```

```
            max = A[i];
```

```
            i = i + 1;
```

```
    }
```

```
    return max;
```

← 2 operações

← 1 operação

← n operações

← 2 operações

← 2 operações

← 2 operações

n - 1
vezes

← 1 operação

Pior caso: $7n - 2$ operações

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

Pior caso: $O(n)$

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

Melhor caso:
Ocorre quando
A[0] é o maior elemento

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

2 operações

1 operação

n operações

2 operações

0 operações

2 operações

n - 1 vezes

1 operação

Melhor caso: A[0] é o maior elemento

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int  
n) {
```

```
    int max = A[0];
```

```
    int i = 1;
```

```
    while (i <= n-1) {
```

```
        if (A[i] > max)
```

```
            max = A[i];
```

```
            i = i + 1;
```

```
    }
```

```
    return max;
```

← 2 operações

← 1 operação

← n operações

← 2 operações

← 0 operações

← 2 operações

n - 1
vezes

← 1 operação

Melhor caso: $2 + 1 + n + 4.(n - 1) + 1$

Análise de Complexidade

Exemplo

```
public static int findMax(int A[], int
n) {
    int max = A[0];
    int i = 1;
    while (i <= n-1) {
        if (A[i] > max)
            max = A[i];
        i = i + 1;
    }
    return max;
}
```

2 operações

1 operação

n operações

2 operações

0 operações

2 operações

n - 1
vezes

1 operação

Melhor caso: 5n operações

Análise de Complexidade

Exemplo

```
for (i = 0; i < n; i++)  
{  
    instruções  
}
```

A contabilização do número de instruções é simples:

n iterações e, em cada uma, são executadas um número constante de instruções

Trecho do Algoritmo: $O(n)$

Análise de Complexidade

Exemplo

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
    {  
        instruções  
    }
```

A contabilização do número de instruções é simples:

O ciclo interno (for j) é $O(n)$ e é executado

Trecho do Algoritmo: $O(n^2)$

Algoritmo de Busca Binária

Análise de Complexidade

O número de iterações realizadas depende do **tamanho do vetor**:

- No início da primeira iteração, o vetor tem tamanho n
- No início da segunda iteração, o vetor passa a ter um tamanho $n/2$
- No início da terceira iteração, o vetor passa a ter um tamanho $n/4$
- E assim sucessivamente ...

No **pior caso**, o vetor atingirá o **tamanho** $= 1$, que ocorre quando $n/2^k = 1$

assim: $n/2^k = 1 \therefore n = 2^k \therefore \log_2 n = k$

Pior caso: $T(n) = O(\log n)$

Melhor caso: quando o número procurado é igual ao elemento do meio na primeira iteração – $T(n) = O(1)$

Exercício (1) notação big O

- Tente demonstrar matematicamente que $n^2 + 800 = O(n^2)$.
- Tente demonstrar matematicamente que $2n + 10 = O(n)$.
- Tente demonstrar matematicamente que $n^2 = O(n)$.
- Tente demonstrar matematicamente que $7n - 2 = O(n)$.
- Tente demonstrar matematicamente que $n^2 + 20n + 5 = O(n^3)$.

Exercício (2) Notação Θ

- Tente demonstrar matematicamente que $n^2+800 = \Theta(n^2)$.
- Tente demonstrar matematicamente que $n+10 = \Theta(n^2)$.
- Tente demonstrar matematicamente que $2n+10 = \Theta(n)$.
- Tente demonstrar matematicamente que $7n-2 = \Theta(1)$.
- Tente demonstrar matematicamente que $n^2+20n+5 = \Theta(n^3)$.

Exercício (3) Notação Ω

- Tente demonstrar matematicamente que $n^2+800 = \Omega(n^2)$.
- Tente demonstrar matematicamente que $n+10 = \Omega(n^2)$.
- Tente demonstrar matematicamente que $2n+10 = \Omega(n)$.
- Tente demonstrar matematicamente que $7n-2 = \Omega(1)$.
- Tente demonstrar matematicamente que $n^2+20n+5 = \Omega(n^3)$.

Conteúdo da Próxima Aula

- Recursividade