EaD
UFMS DIGITAL
Graduação

AGEAD
Agência de Educação
Digital e a Distância

UFMS

Algoritmos e Programação II

Prof. Dr. Rafael dos Passos Canteri

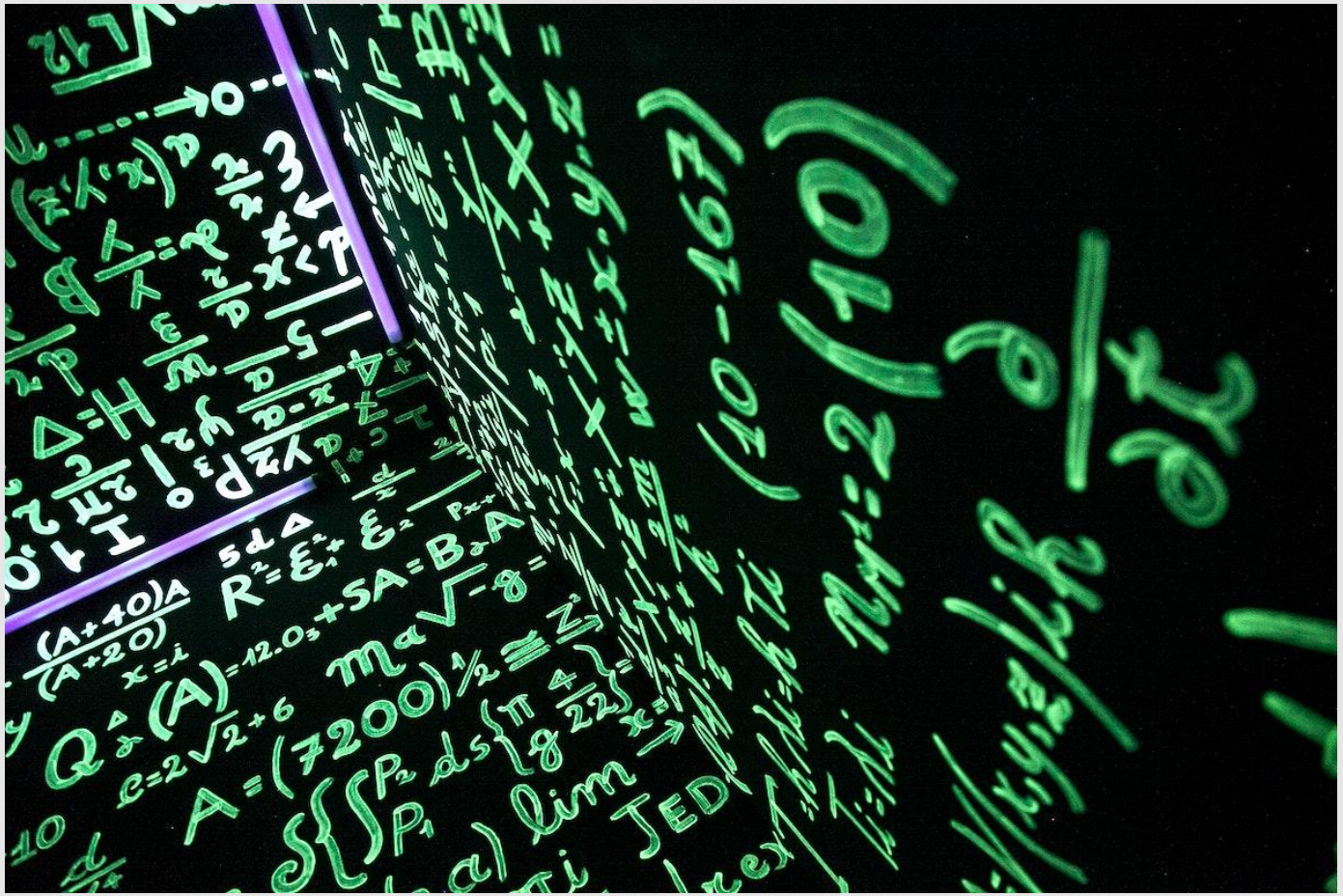
EaD
UFMS Digital
Graduação

Módulo 4 - Noções de eficiência e recursividade

Unidade 1 - Análise de complexidade de algoritmos

UFMS
AGEAD
Agência de Educação Digital e a Distância

Complexidade de algoritmos



Fonte: Qubitor

Análise de algoritmos

- Algoritmos distintos podem resolver um mesmo problema, mas não com a mesma eficiência.
- As diferenças de desempenho entre algoritmos podem:
  - Ser insignificantes em entradas pequenas,
  - mas podem se tornar drásticas com o aumento do número de elementos.

Análise de algoritmos

- A subárea da Computação responsável por avaliar esse desempenho é chamada de Complexidade Computacional, que estuda o custo de execução de algoritmos, geralmente em termos de:
  - Tempo de processamento
  - Memória utilizada

Custo total de um algoritmo = tempo + memória

Perguntas da análise de algoritmos

- Existe um algoritmo que resolva o problema de forma eficiente e realista?
- Quais são os recursos computacionais exigidos por esse algoritmo?
- Existe um algoritmo melhor que execute a mesma tarefa?
- Como comparar algoritmos de forma objetiva e sistemática?

Importância da análise de algoritmos

- Permite o desenvolvimento de soluções escaláveis e com melhor desempenho.
- Evita desperdício de recursos computacionais.
- Ajuda a compreender os limites da computação e o que é ou não viável computacionalmente.
- Melhora o processo de tomada de decisões de projeto na construção de sistemas.

Importância da análise de algoritmos

- Embora seja possível analisar a complexidade após a implementação, o ideal é considerar critérios de desempenho desde o início do projeto.
- Isso ajuda a criar algoritmos não apenas corretos, mas também eficientes e sustentáveis para aplicações reais.

Análise matemática de algoritmos

- Estudo formal do algoritmo.
- Avalia a complexidade do algoritmo independentemente do *hardware*.
- Independe da linguagem de programação.
- Faz simplificações e considera apenas os custos dominantes do algoritmo.
- Permite entender o funcionamento do algoritmo em função dos dados de entrada.

Análise assintótica

- Foco no crescimento: considera-se apenas o comportamento dominante conforme o tamanho da entrada aumenta.
- Ignora detalhes menores: termos de crescimento lento e constantes são descartados.
- Independência da linguagem: a análise não depende da linguagem de programação utilizada.
- Entrada crítica: assume-se uma entrada principal com tamanho N, como o número de elementos de um vetor.

Análise assintótica

- Notação Big-O: Utiliza-se a notação O(grande) para representar a ordem de crescimento da complexidade.
- Quanto maior a classe de complexidade, menor a eficiência do algoritmo.

Classes de Complexidade

1. Complexidade Constante
2. Complexidade Logarítmica
3. Complexidade Linear
4. Complexidade Linearítmica
5. Complexidade Quadrática
6. Complexidade Cúbica
7. Complexidade Exponencial
8. Complexidade Fatorial

Classes de Complexidade

Complexidade	Nome	Exemplo de Algoritmo	Escalabilidade
O(1)	Constante	Acesso direto ao vetor	Altamente eficiente
O(logn)	Logarítmica	Busca binária	Muito eficiente
O(n)	Linear	Busca linear	Escala proporcional
O(nlogn)	Linearítmica	Merge sort, Quick sort (caso médio)	Boa escalabilidade

Classes de Complexidade

Complexidade	Nome	Exemplo de Algoritmo	Escalabilidade
$O(n^2)$	Quadrática	Bubble sort, Insertion sort	Pouco eficiente
$O(n^3)$	Cúbica	Produto de matrizes	Pouco eficiente
$O(2^n)$	Exponencial	Algoritmos de força bruta	Altamente ineficiente
$O(n!)$	Fatorial	Algoritmos de permutação total	Inviável

Exemplos em código

- Complexidade linear (n).

```
def busca(valor, vetor):
    for i in range(len(vetor)):
        if valor == vetor[i]:
            return i
    return -1
```

Exemplos em código

- Complexidade quadrática (n^2).

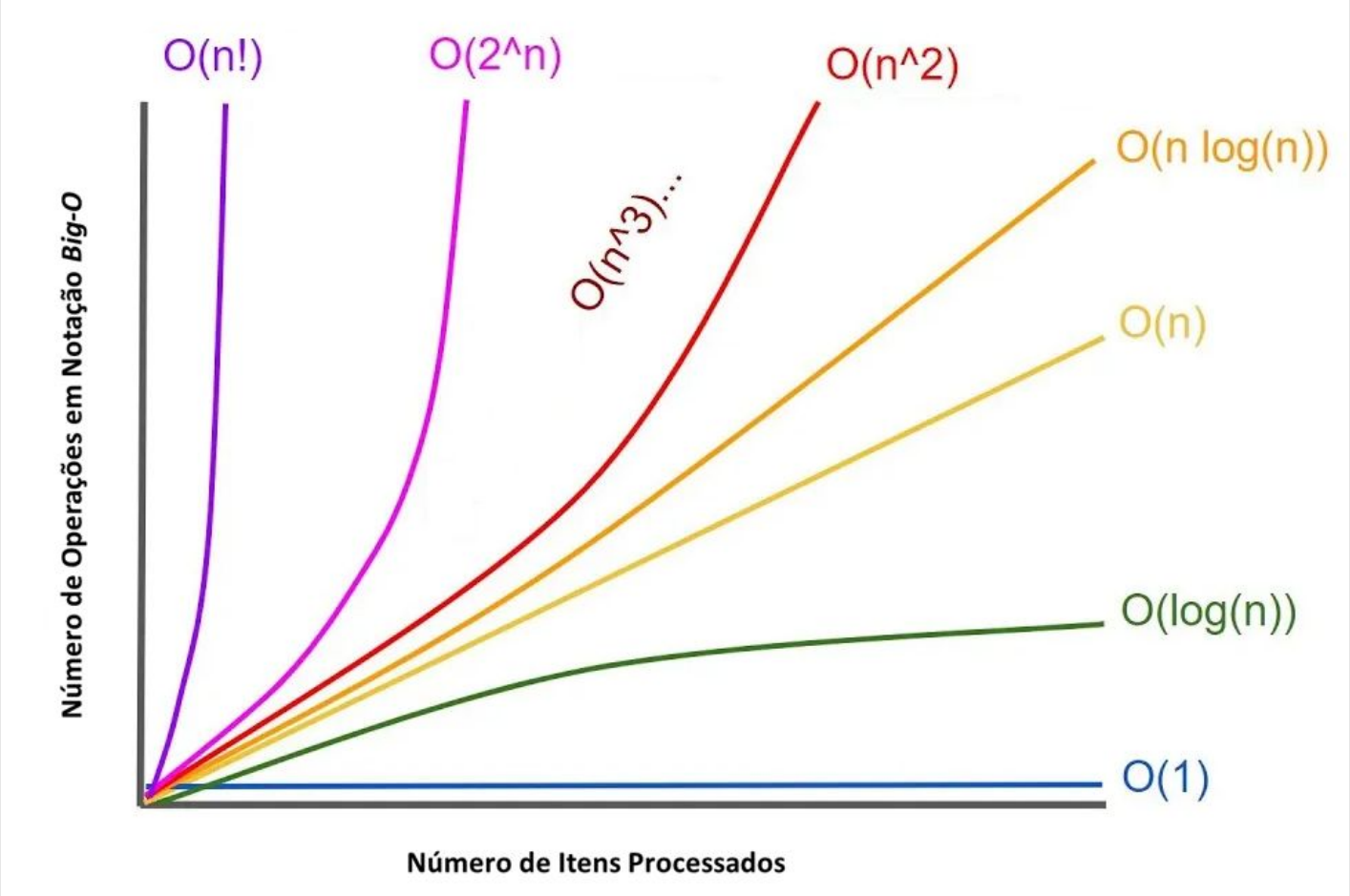
```
def imprime_matriz(mat):
    for i in range(n):
        for j in range(n):
            print(f"{mat[i][j]}", end=" ")
        print()
```

Exemplos em código

- Complexidade cúbica (n^3).

```
def altera_matriz(mat):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                mat[i][j][k] = -1 * mat[i][j][k]
```

Comparação de complexidades



Fonte: Traduzido de Stackademic

Recapitulação

- A importância de analisar algoritmos além da implementação.
- Como a análise assintótica nos ajuda a prever desempenho.
- As principais classes de complexidade Big-O e suas implicações práticas.
- A diferença entre algoritmos eficientes e ineficientes em escalabilidade.

Referências

QUBITOR. Qubitor. [S.d.]. Disponível em: https://link.ufms.br/Vvlqj. Acesso em: 9 jun. 2025.

SALVI, Priya. **How to calculate Big O notation time complexity**. Stackademic, 20 ago. 2023. Disponível em: https://link.ufms.br/waKIW. Acesso em: 9 jun. 2025.

SZWARCFITER, Jayme L.; MARKENZON, Lilian. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2010. ISBN 978-85-216-2995-5.

Licenciamento

Respeitadas as formas de citação formal de autores de acordo com as normas da ABNT NBR 6023 (2018), a não ser que esteja indicado de outra forma, todo material desta apresentação está licenciado sob uma Licença Creative Commons - Atribuição 4.0 Internacional.

EaD

UFMS DIGITAL

Graduação