

Aula 04

Vetores Ordenados - Busca Binária



Algoritmos e Estrutura de Dados II

2º Semestre - CDN



Prof. Dr. Dilermando Piva Jr.

Conteúdo Programático - Planejamento

| Semana | Data | Temas/Atividades |
|--------|-------|---|
| 1 | 07/08 | Acolhimento e Boas-vindas! Introdução a Disciplina. Formas de Avaliação e Percurso Pedagógico. |
| 2 | 14/08 | Tipo de dado abstrato. Introdução a Estrutura de Dados. |
| 3 | 21/08 | Complexidade de Algoritmos |
| 4 | 28/08 | Vetores não-Ordenados e busca sequencial |
| 5 | 04/09 | Vetores Ordenados e busca binária |
| 6 | 11/09 | Revisão de Programação Orientada a Objetos (POO) |
| 7 | 18/09 | Pilhas |
| 8 | 25/09 | Filas |
| 9 | 02/10 | Listas encadeadas |
| 10 | 09/10 | Recursão |
| 11 | 16/10 | Primeira Avaliação Formal. (P1). Correção da Avaliação após o intervalo. |
| 12 | 18/10 | Algoritmos de Ordenação |
| 13 | 23/10 | Algoritmos de Ordenação |
| 14 | 30/10 | Árvores |
| 15 | 06/11 | Grafos |
| 16 | 13/11 | Segunda Avaliação Formal (P2). Correção da Avaliação após o intervalo |
| 17 | 27/11 | Apresentação PI do curso de CDN |
| 18 | 04/12 | Tabela Hash (tabela de espalhamento) – Tópico extra. |
| 19 | 11/12 | Exame / Avaliação Substitutiva. Correção da Avaliação após o intervalo. Finalização Disciplina |
| 20 | 18/12 | Finalização da disciplina. |

Cenário:

Catálogo de produtos
(ordenado por nome)

Pular
diretamente para
a parte do
catálogo próxima
ao item desejado

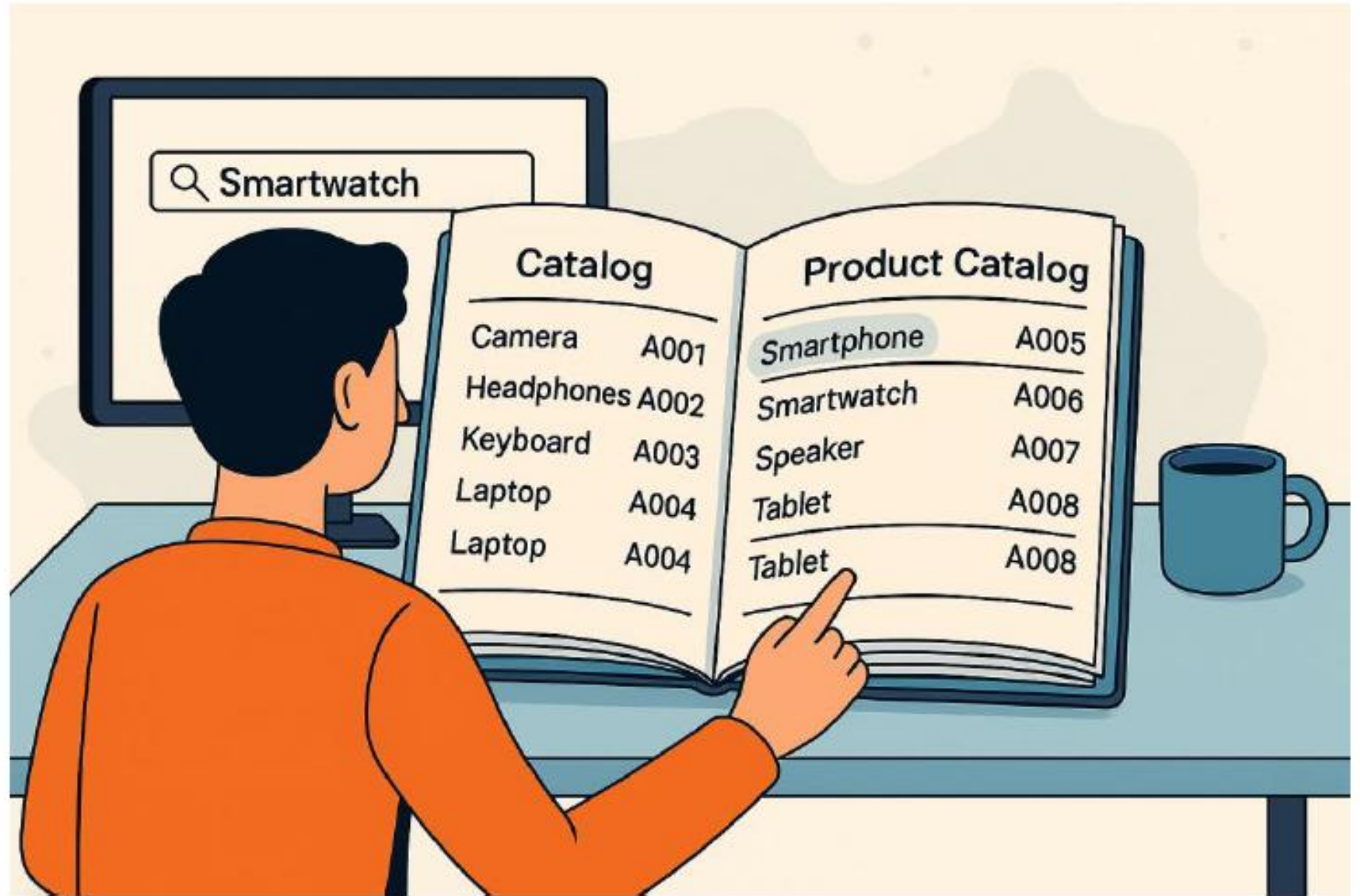


Figura: Busca por um produto em um catálogo ordenado por nome

Algoritmos de Pesquisa

Estruturas ordenadas - Busca binária

Busca em uma lista telefônica...



Busca página por página.



LIVRO / LISTA TELEFÔNICA: \rightarrow 1000 páginas (n)

Melhor dos casos: 1 página

Pior dos casos: 1000 páginas

Média: 500 páginas

Complexidade: $O(n) \rightarrow O(1000)$

sempre pega o pior caso.

Busca binária.

Número de verificações será **logaritmo**



LIVRO / LISTA TELEFÔNICA: \rightarrow 1000 páginas (n)

Melhor dos casos: 1 página

Pior dos casos: $\log_2 1000 \approx 10$

Média: 5 páginas

Complexidade: $O(\log n) \rightarrow O(10)$
sempre pega o pior caso.

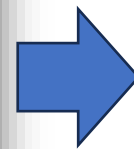
Busca binária.

Número de verificações será **logaritmo**
(como?)



Número de Comparações

- Inicialmente, temos n elementos.
- Após 1 comparação, reduzimos para $\frac{n}{2}$.
- Após 2 comparações, reduzimos para $\frac{n}{4}$.
- Após 3 comparações, reduzimos para $\frac{n}{8}$.



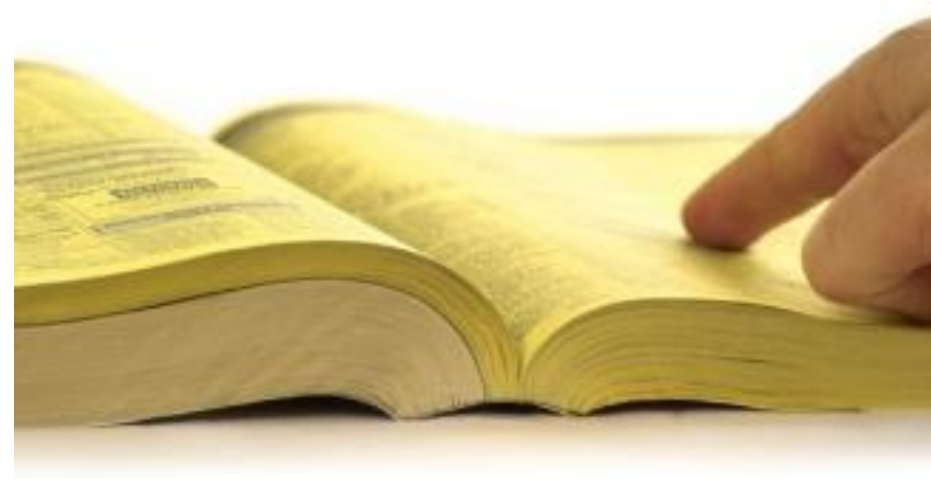
Após k comparações, o tamanho da lista restante será:

$$\frac{n}{2^k}$$

Busca binária.

Número de verificações será **logaritmo**

(como?)



A busca termina quando sobra apenas um elemento para ser comparado, ou seja:

$$\frac{n}{2^k} = 1$$

Agora, resolvendo para k :

$$n = 2^k$$

Aplicamos logaritmo base 2 nos dois lados:

$$\log_2(n) = k$$

Portanto, o número máximo de comparações necessárias para encontrar um elemento (ou determinar que ele não existe) é aproximadamente:

$$O(\log_2 n)$$

Atividade com IA

- Se ainda tiver dúvida...:
 - Como chegou a $O(\log n)$ no caso de busca em vetores ordenados?



- Faça individualmente, e depois compartilhe com o seu colega esses conceitos.

Se coloque como um professor de matemática, com profundos conhecimentos em ciência da computação. Explique de forma mais detalhada possível, para um adolescente que está no primeiro ano do ensino médio, a razão pela qual a quantidade máxima de comparações utilizando busca binária é de, no máximo, “log n”.

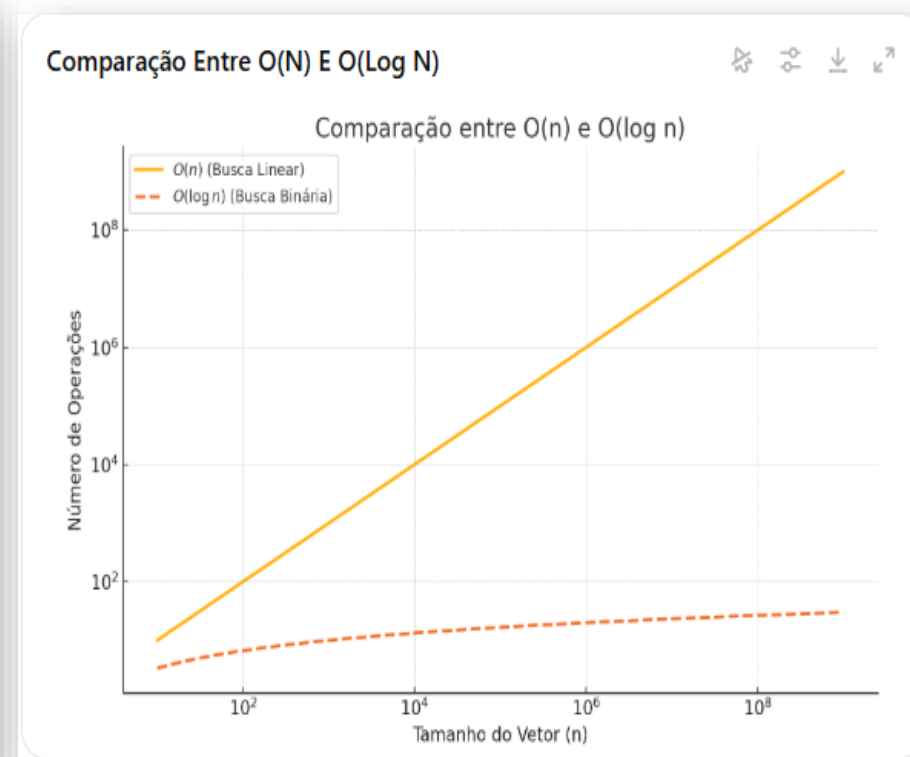
Busca binária.

Número de verificações será **logaritmo**

(Comparação...)

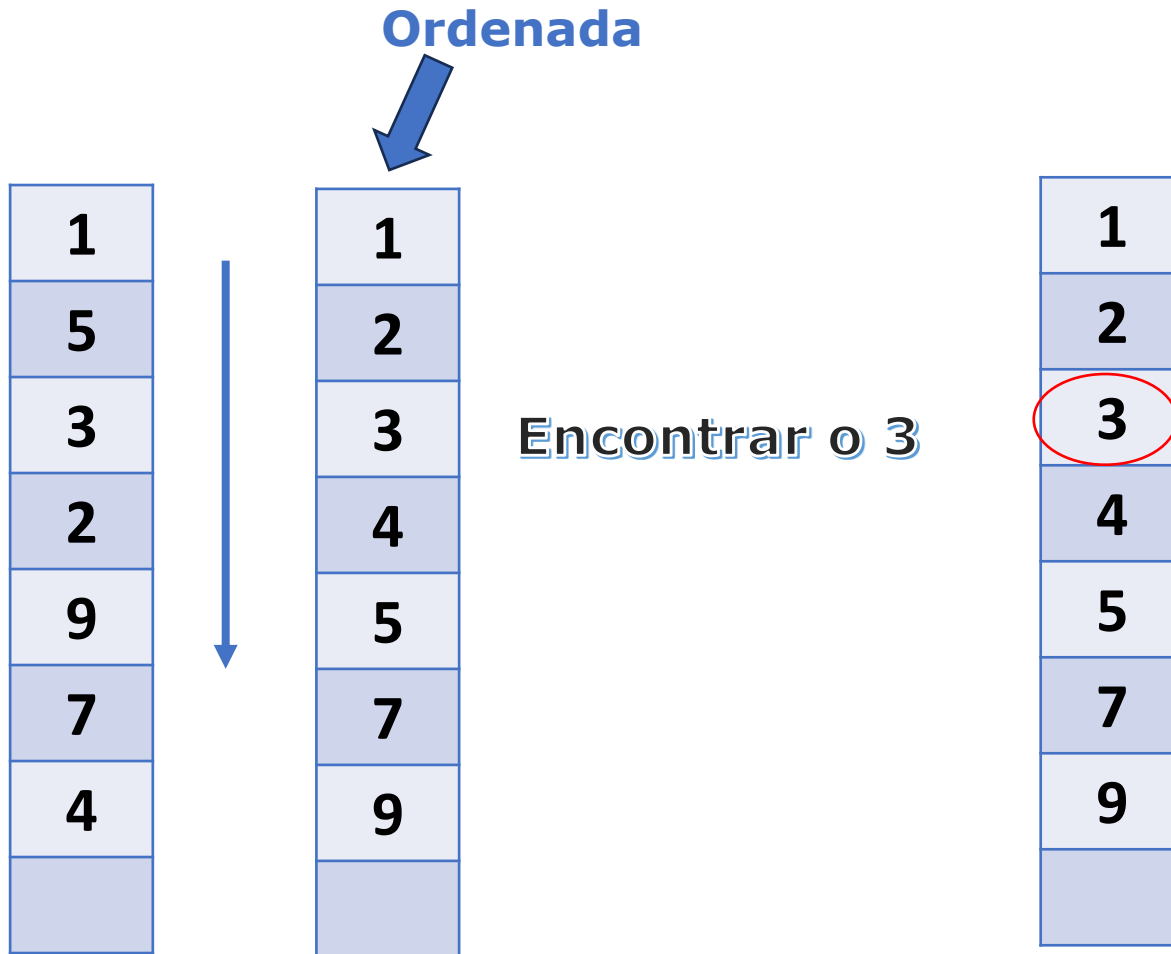


| Número de Elementos (n) | Busca Linear ($O(n)$) | Busca Binária ($O(\log_2 n)$) |
|--------------------------------|-------------------------|---------------------------------|
| 10 | 10 | 4 |
| 100 | 100 | 7 |
| 1.000 | 1.000 | 10 |
| 1.000.000 | 1.000.000 | 20 |
| 1.000.000.000 | 1.000.000.000 | 30 |



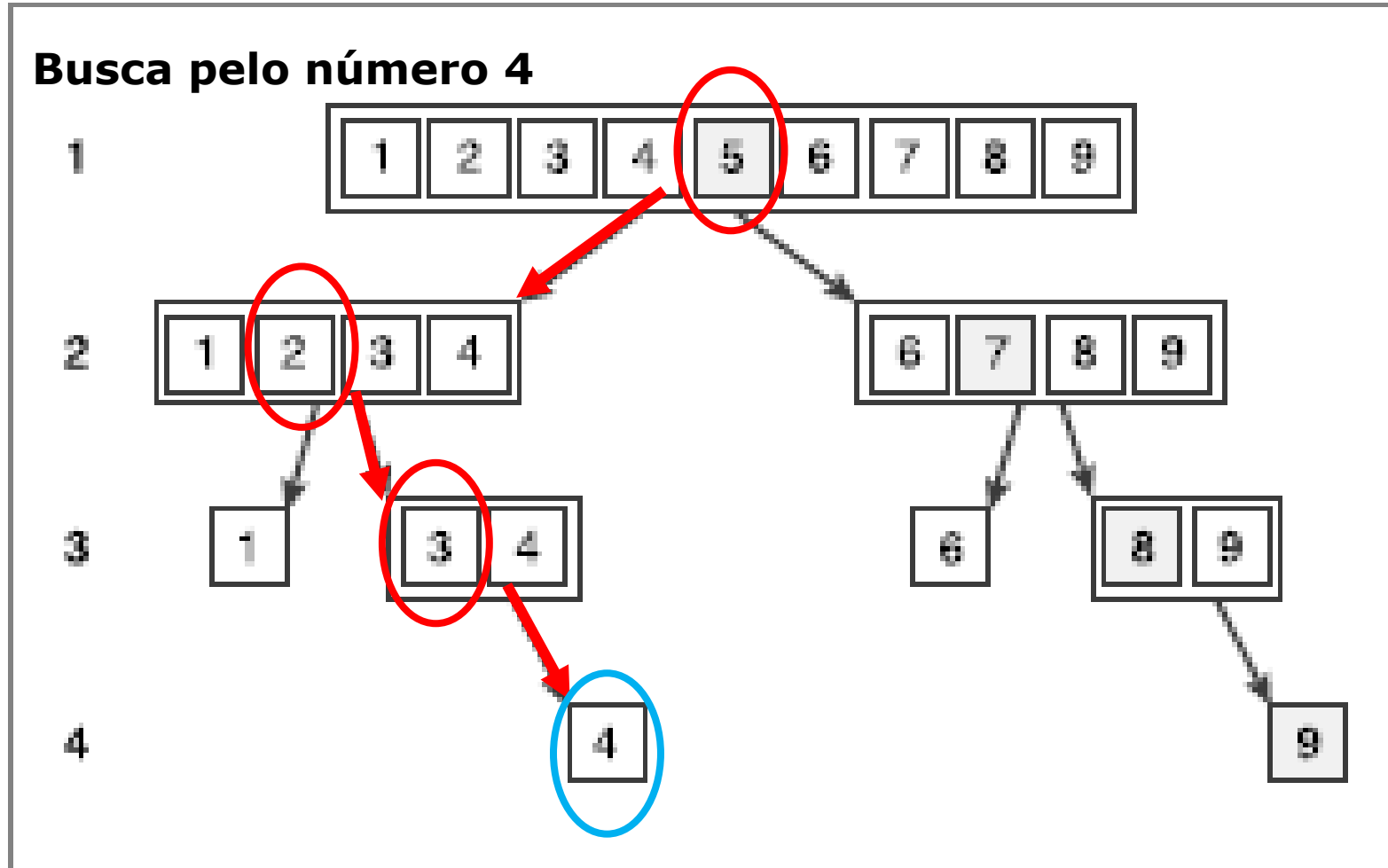
Busca em uma Lista Ordenada

- **Problema:** buscar um valor v existente em uma lista sequencial ordenada com N elementos
 - o tempo de busca pode ser reduzido significativamente
 - devido a relação entre os elementos da lista
 - mesmo para um conjunto de elementos é grande



Busca Binária em uma Lista Ordenada

- **Problema:** buscar um valor v não existente em uma lista sequencial ordenada com N elementos
 - o tempo de busca pode ser reduzido significativamente
 - devido a relação entre os elementos da lista
 - mesmo para um conjunto de elementos é grande



Pesquisa binária em uma lista ordenada (1 de 2)

Código para a função de pesquisa binária:

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1  
        else:  
            esquerda = meio + 1  
    return -1
```

Atividade com IA

- Algoritmo de Pesquisa Binária...
 - Como funciona na prática?



- Faça individualmente, e depois compartilhe com o seu colega esses conceitos.

Pergunte a alguma IA como o algoritmo de busca binária funciona ... Fale para ela explicar detalhadamente...

Exemplo de prompt

Tomando como base o seguinte algoritmo:

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1  
        else:  
            esquerda = meio + 1  
    return -1
```

Persona: Um professor de matemática com profundos conhecimentos em ciência da computação. O tom deve ser didático, claro e encorajador.

Contexto: A explicação é para um estudante do primeiro ano do ensino médio que está aprendendo sobre algoritmos e notação Big O. Ele já sabe o que é um algoritmo e a importância da eficiência. O estudante forneceu um código Python (acima) do algoritmo de busca binária e precisa que você o explique linha por linha.

Problema: Explique o funcionamento do código Python do algoritmo de busca binária, detalhando o propósito de cada linha.

Forma de Solução:

- Apresente o código fornecido pelo estudante.
- Divida a explicação em seções lógicas, com títulos claros, como "Preparando o Terreno", "O Ciclo da Busca" e "As Três Possibilidades".
- Para cada seção, copie as linhas de código relevantes e explique o seu propósito.
- Use um exemplo concreto e numérico (ex: buscar o número 13 em uma lista de 16 elementos) para ilustrar como as variáveis (esquerda, direita, meio) mudam a cada iteração.
- Mantenha a linguagem simples e acessível, fazendo analogias e perguntas retóricas para facilitar o entendimento.
- Ao final, conclua a explicação reafirmando a lógica do algoritmo e sua eficiência.

Pesquisa binária em uma lista ordenada (1 de 2)

Código para a função de pesquisa binária:

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1  
        else:  
            esquerda = meio + 1  
    return -1
```

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):
```

ASSINATURA DA FUNÇÃO

- `v`: O valor que estamos procurando.
- `lista_ord`: O vetor **ordenado** onde será feita a busca.

💡 **Importante:** A busca binária só funciona em listas ordenadas. Se a lista não estiver ordenada, o algoritmo não funciona corretamente.

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):
```

```
    esquerda = 0
```

```
    direita = len(lista_ord) - 1
```

INICIALIZAÇÃO DAS VARIÁVEIS

- `esquerda` → Marca o início da parte do vetor que estamos analisando (começa no índice `0`).
- `direita` → Marca o fim da parte do vetor que estamos analisando (começa no índice `len(lista_ord) - 1`, ou seja, o último índice).

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:
```

LOOP PRINCIPAL (BUSCA BINÁRIA)

O loop continua enquanto a parte do vetor que estamos analisando for válida (`esquerda` ainda não passou `direita`).

Cada iteração reduz pela metade a quantidade de elementos que precisamos procurar.

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2
```

ENCONTRANDO O MEIO

Aqui, calculamos o índice do meio do vetor.

💡 Por que `// 2` ?

Isso garante que sempre pegamos um índice válido (divisão inteira).

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio
```

COMPARAÇÃO COM O VALOR PROCURADO

- Se o elemento do meio (`lista_ord[meio]`) for igual a `v`, retornamos o índice e terminamos a busca.
- Se encontramos o elemento, não precisamos continuar.

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1
```

AJUSTANDO A REGIÃO DE BUSCA (caso o valor v não esteja no meio)

Se v for menor que o valor no meio, então ele só pode estar na metade esquerda.

Portanto, ajustamos `direita` para `meio - 1`.

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1
```

else:

esquerda = meio + 1


O VALOR ESTÁ DEPOIS DO MEIO

Se `v` for maior que o valor no meio, então ele só pode estar na metade direita.

Portanto, ajustamos `esquerda` para `meio + 1`.

Pesquisa binária em uma lista ordenada (explicação)

```
def busca_binaria(v, lista_ord):  
    esquerda = 0  
    direita = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1  
        else:  
            esquerda = meio + 1
```



return -1

O ELEMENTO NÃO FOI ENCONTRADO

Se `esquerda` ultrapassar `direita`, significa que o elemento não está na lista, então retornamos `-1` como um indicativo de falha.

Um exemplo PRÁTICO

```
lista = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
v = 7
```

```
resultado = busca_binaria(v, lista)
print(resultado)
```

Passo a Passo...

| Iteração | esquerda | direita | meio | lista_ord[meio] | Comparação |
|----------|----------|---------|------|-----------------|---|
| 1º | 0 | 9 | 4 | 9 | $7 < 9$, então <code>direita = 3</code> |
| 2º | 0 | 3 | 1 | 3 | $7 > 3$, então <code>esquerda = 2</code> |
| 3º | 2 | 3 | 2 | 5 | $7 > 5$, então <code>esquerda = 3</code> |
| 4º | 3 | 3 | 3 | 7 | $7 == 7$, então retorna 3 |

📌 O valor 7 foi encontrado na posição 3.

Pesquisa binária em uma lista ordenada (1 de 2)

Código para a função de pesquisa binária:

```
def busca_binaria(v, lista_ord):  
    esquerda = 0
```

<https://www.cs.usfca.edu/~galles/visualization/Search.html>

```
        meio = (esquerda + direita) // 2  
        elif v < lista_ord[meio]:  
            direita = meio - 1  
        else:  
            esquerda = meio + 1  
    return -1
```

Busca Binária... Complexidade

- Número de passos:
 - suponha que $n = 2^k$ sem perda de generalidade
 - 1o passo: uma lista de **n** elementos = $n/2^0$ elementos
 - 2o passo: uma lista de **n/2** elementos = $n/2^1$ elementos
 - 3o passo: uma lista de **n/4** elementos = $n/2^2$ elementos
 - k-ésimo passo: uma lista de **n/2^(k-1)** elementos
 - (k+1)-ésimo passo: uma lista de **n/2^k** elementos
 - sendo o último passo, resta na lista somente 1 elemento
- Logo:
 - $n/2^k = 1 \rightarrow k = \mathbf{\log_2 n}$

Pesquisa binária em uma lista ordenada (1 de 2)

Código para a função de pesquisa binária:

```
def busca_binaria(v, lista_ord):  
    direita = 0  
    esquerda = len(lista_ord) - 1  
    while esquerda ≤ direita:  
        meio = (esquerda + direita) // 2  
        if v == lista_ord[meio]:  
            return meio  
        elif v < lista_ord[meio]:  
            direita = meio - 1  
        else:  
            esquerda = meio + 1  
    return -1
```

Complexidade?

$O(\log n)$

VAMOS PARA A PRÁTICA ?!!!



Exercícios...



1.(ENADE): A **busca binária** possui complexidade de tempo diferente da busca sequencial. Suponha um vetor de tamanho n já ordenado. Qual das alternativas descreve **corretamente** a complexidade de tempo da busca binária, em termos de melhor caso e pior caso, comparando-a à busca sequencial?

- A. Melhor caso $O(n)$ e pior caso $O(n^2)$ na busca binária, o que é mais eficiente que a busca sequencial.
- B. Melhor caso $O(1)$ e pior caso $O(\log n)$ na busca binária, enquanto a busca sequencial tem melhor caso $O(1)$ e pior caso $O(n)$.
- C. Melhor caso e pior caso $O(\log n)$ na busca binária, pois sempre dividimos a busca pela metade; a busca sequencial sempre é $O(n)$ em qualquer caso.
- D. Melhor caso $O(1)$ e pior caso $O(n)$ tanto para busca binária quanto para busca sequencial.

Exercícios...



1.(ENADE): A **busca binária** possui complexidade de tempo diferente da busca sequencial. Suponha um vetor de tamanho n já ordenado. Qual das alternativas descreve **corretamente** a complexidade de tempo da busca binária, em termos de melhor caso e pior caso, comparando-a à busca sequencial?

- A. Melhor caso $O(n)$ e pior caso $O(n^2)$ na busca binária, o que é mais eficiente que a busca sequencial.
- B. Melhor caso $O(1)$ e pior caso $O(\log n)$ na busca binária, enquanto a busca sequencial tem melhor caso $O(1)$ e pior caso $O(n)$.
- C. Melhor caso e pior caso $O(\log n)$ na busca binária, pois sempre dividimos a busca pela metade; a busca sequencial sempre é $O(n)$ em qualquer caso.
- D. Melhor caso $O(1)$ e pior caso $O(n)$ tanto para busca binária quanto para busca sequencial.

Exercícios...



2.(TEÓRICA): Por que o algoritmo de **busca binária exige que o vetor esteja ordenado** previamente? O que poderia acontecer se tentássemos aplicar a busca binária em um conjunto de dados não ordenado?

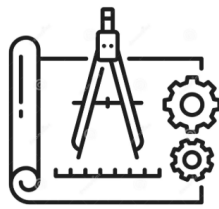
Exercícios...



2.(TEÓRICA): Por que o algoritmo de **busca binária exige que o vetor esteja ordenado** previamente? O que poderia acontecer se tentássemos aplicar a busca binária em um conjunto de dados não ordenado?

Solução: A busca binária baseia-se na premissa de que o conjunto de dados está ordenado para poder descartar metade dos elementos a cada comparação. É graças à ordem que, ao comparar com o elemento do meio, sabemos em qual metade o valor buscado pode estar (se é menor, só pode estar à esquerda; se é maior, só pode estar à direita). Se o vetor não estiver ordenado, essa lógica não funciona – o elemento do meio não fornece informação confiável sobre a posição do alvo. Por exemplo, imagine um vetor não ordenado: ao comparar um certo valor com um elemento "do meio", não há garantia de que todos os elementos à esquerda sejam menores ou maiores de forma consistente. Assim, poderíamos descartar metade dos dados erradamente e acabar ignorando a parte onde o valor buscado realmente está. Na prática, aplicar busca binária em dados não ordenados resultaria em falhas na localização do elemento (o algoritmo poderia concluir que o elemento não existe quando na verdade está presente, porém fora do lugar esperado). Portanto, a ordenação prévia é uma condição indispensável para a correção da busca binária. (Em suma: sem ordem, não há “divisão” coerente do espaço de busca.)

Projeto...

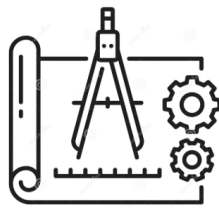


Busca Eficiente em um Índice de Livros

Contexto: Você está trabalhando em um projeto para gerenciar uma biblioteca. As informações sobre os livros (**título, autor, ano de publicação**) estão armazenadas em uma lista, e essa lista é mantida ordenada alfabeticamente pelo título do livro.

Problema: Os usuários precisam encontrar rapidamente um livro específico sabendo seu título. Uma busca ineficiente pode dificultar a localização dos livros desejados.

Projeto...



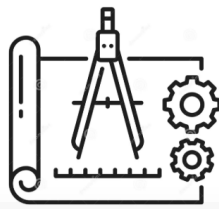
Busca Eficiente em um Índice de Livros

Contexto: Você está trabalhando em um projeto para gerenciar uma biblioteca. As informações sobre os livros (título, autor, ano de publicação) estão armazenadas em uma lista, e essa lista é mantida ordenada alfabeticamente pelo título do livro.

Problema: Os usuários precisam encontrar rapidamente um livro específico sabendo seu título. Uma busca ineficiente pode dificultar a localização dos livros desejados.

Indicação de como resolver: *Desenvolva uma funcionalidade que permita aos usuários buscar um livro pelo título utilizando a busca binária. Ao receber o título do livro, sua função deve procurar no índice ordenado. Se o livro for encontrado, exiba suas informações (título, autor, ano). Caso contrário, informe que o livro não foi encontrado.*

Projeto...



Busca Eficiente em um Índice de Livros

```
def buscar_livro_indice(indice_livros, titulo_livro):
    baixo = 0
    alto = len(indice_livros) - 1
    while baixo <= alto:
        meio = (baixo + alto) // 2
        if indice_livros[meio]['titulo'].lower() == titulo_livro.lower():
            return indice_livros[meio]
        elif indice_livros[meio]['titulo'].lower() < titulo_livro.lower():
            baixo = meio + 1
        else:
            alto = meio - 1
    return None

# Exemplo de índice de livros (já ordenado alfabeticamente por título)
indice_biblioteca =

titulo_busca = input("Digite o título do livro que você procura: ")
livro_encontrado = buscar_livro_indice(indice_biblioteca, titulo_busca)

if livro_encontrado:
    print(f"Livro encontrado: {livro_encontrado['titulo']}")
    print(f"Autor: {livro_encontrado['autor']}")
    print(f"Ano de publicação: {livro_encontrado['ano']}")
else:
    print(f"O livro com o título '{titulo_busca}' não foi encontrado na biblioteca.")
```