# ‹Suffix Array› + ‹LCP›

**Kaio Christaldo**
**Fabricio Matsunaga**

# ‹Suffix Array›

# Apresentação Problema Motivador

## 🧬 Enunciado: Fitas de DNA - Contagem de Padrões

O professor André, da disciplina de Biologia Computacional, está analisando uma longa fita de DNA, representada por uma string que contém apenas os caracteres `A`, `C`, `G` e `T`.

Ele quer que seus alunos desenvolvam um programa que o ajude a contar quantas vezes determinadas **sequências genéticas** (padrões) aparecem **como substrings** na fita de DNA.

Você deve escrever um programa que, dada a fita de DNA e uma lista de sequências genéticas, diga para cada uma **quantas vezes ela aparece como substring** na fita.

A entrada contém:

- Uma string `S` ($1 \leq |S| \leq 10^5$) representando a fita de DNA (apenas letras `A`, `C`, `G`, `T`).
- Um inteiro `Q` ($1 \leq Q \leq 10^4$), o número de sequências genéticas a serem testadas.
- Em seguida, `Q` linhas contendo cada uma uma string $P_i$ ($1 \leq |P_i| \leq 100$), representando o padrão genético a ser buscado.

Para cada padrão $P_i$, imprima uma linha contendo um inteiro: o número de vezes que ele aparece como substring em `S`.

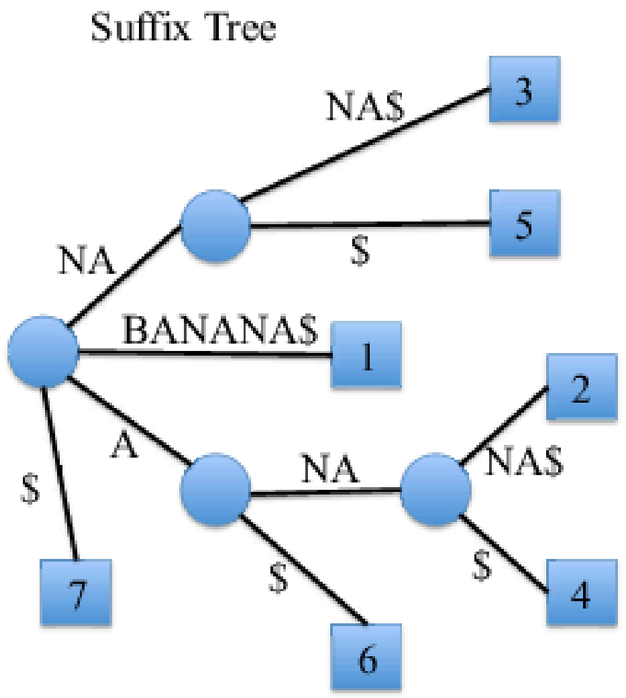
# Algorithm ‹suffix array›

- O Suffix Array é uma técnica utilizada para buscas eficientes em strings. Basicamente, ele consiste em construir um vetor ordenado com todas as posições iniciais dos sufixos de uma dada string, permitindo assim consultas rápidas sobre substrings.
- O Suffix Array foi introduzido por **Udi Manber** e **Gene Myers** em 1990.
- Eles propuseram o Suffix Array como uma alternativa mais simples e eficiente em termos de memória ao Suffix Tree, para resolver problemas de busca em strings.
  - Suffix Tree: mais antigo, complexo e pesado em memória.
  - Suffix Array: mais compacto, ordena os sufixos de uma string em ordem lexicográfica, permitindo buscas eficientes com técnicas como binary search.

# Algorithm ‹suffix array›

T=BANANA$

| *i* | SA | LCP | suffix |
|---|---|---|---|
| 1 | 7 | -1 | $ |
| 2 | 6 | 0 | A$ |
| 3 | 4 | 1 | ANA$ |
| 4 | 2 | 3 | ANANA$ |
| 5 | 1 | 0 | BANANA$ |
| 6 | 5 | 0 | NA$ |
| 7 | 3 | 2 | NANA$ |




## suffix array vs suffix tree

- Uma matriz de sufixos e a árvore de sufixos podem ser construídas uma a partir da outra em tempo linear.

# Algorithm <suffix array>

## Ideia:

```
Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ---------------->     1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

## Implementações no Drive:

[Drive: Suffix Array]

# Algorithm <suffix array>

## Suffix tree

| Operação | Complexidade |
| --- | --- |
| Construção | O(n) (com algoritmos como Ukkonen, McCreight, etc.) |
| Busca de substring (pattern de tamanho m) | O(m) |
| Espaço | O(n) (mas com grande constante) |

# Algorithm ‹suffix array›

## Suffix array

| Operação | Complexidade |
| --- | --- |
| Construção (algoritmos rápidos) | O(n log n), ou até O(n) com algoritmos sofisticados |
| Busca de substring (com Binary Search) | O(m log n) |
| Espaço | O(n) |

# Algorithm ‹suffix array›

## Suffix Tree

- Muito eficiente, mas difícil de implementar corretamente.
- Ocupa muito espaço (muitos ponteiros e strings), e consome muita memória na prática.
- Algoritmos lineares como o de Ukkonen (1995) constroem em tempo linear, mas são difíceis de programar.

## Suffix Array

- Mais fácil de implementar.
- Pode ser combinado com o LCP Array para responder perguntas como "maior substring comum".
- Algoritmos lineares de construção: SA-IS, Skew Algorithm, DC3, etc.

# Resolução do Problema Motivador

## k0001 - Fitas de DNA

**Dicas:**

utilizar lower_bound e upper_bound.

| Entradas: 1 | Saídas: 1 |
| --- | --- |
| GGGAAAGGGAAA | 2 |
| 4 | 2 |
| GGG | 2 |
| AAA | 0 |
| GGA | |
| T | |
| --- | --- |
| Entradas: 2 | Saídas: 2 |
| ACGTACGTGACG | 3 |
| 3 | 1 |
| ACG | 1 |
| TGA | |
| GAC | |

**A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução!** 😊

# Apresentação Problema Motivador

## 14. Longest Common Prefix

Easy | Topics | Companies

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string `""`.

**Example 1:**

```
Input: strs = ["flower","flow","flight"]
Output: "fl"
```

**Example 2:**

```
Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.
```

**Constraints:**

- `1 <= strs.length <= 200`
- `0 <= strs[i].length <= 200`
- `strs[i]` consists of only lowercase English letters if it is non-empty.

[1237 - Comparação de Substring]


LCP

# ‹LCP – Longest Common Prefix›

Dado um array de strings arr[], a tarefa é retornar o maior prefixo comum entre cada uma das strings presentes dentro do array. Se não houver nenhum prefixo comum em todas as strings, retorne “ “.

# ‹LCP – Longest Common Prefix›

**Example 1:**
**Input: strs = ["flower","flow","flight"]**
**Output: "fl"**

**Example 2:**
**Input: strs = ["dog","racecar","car"]**
**Output: ""**

## ‹LCP – Força Bruta ›

1. Nos começamos com a primeira string e comparamos seus caracteres com os caracteres corespondentes de todas as outras strings
2. Usaremos dois loop aninhados, com o loop externo iterando sobre os caracteres da primeira string, e o loop interno iterando sobre as outras strings.
3. O prefixo comum é encontrado quando um caracter é diferente ou o quando chegamos no final de uma string.

Complexidade de tempo: O(N*M), onde N é o numero de strings e M é o tamanho da menor string.

# ‹LCP – Força Bruta ›

**01** Step | All the characters in the first iteration (0th index) are same, i.e., 'g'. So append it to the result.

Result = “g”

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr[0] = | g | e | e | k | s | f | o | r | g | e | e | k | s |
| arr[1] = | g | e | e | k | s | | | | | | | | |
| arr[2] = | g | e | e | k | | | | | | | | | |
| arr[3] = | g | e | e | z | e | r | | | | | | | |

Longest Common Prefix using Character by Character Matching

# ‹LCP – Força Bruta ›

**02** Step | All the characters in the second iteration (1st index) are same, i.e., 'e'. So append it to the result.

Result = “ge”

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr[0] = | g | e | e | k | s | f | o | r | g | e | e | k | s |
| arr[1] = | g | e | e | k | s | | | | | | | | |
| arr[2] = | g | e | e | k | | | | | | | | | |
| arr[3] = | g | e | e | z | e | r | | | | | | | |

Longest Common Prefix using Character by Character Matching

# ‹LCP – Força Bruta ›

**03** Step | All the characters in the third iteration (2nd index) are same, i.e., 'e'. So append it to the result.

Result = “gee”

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr[0] = | g | e | e | k | s | f | o | r | g | e | e | k | s |
| arr[1] = | g | e | e | k | s | | | | | | | | |
| arr[2] = | g | e | e | k | | | | | | | | | |
| arr[3] = | g | e | e | z | e | r | | | | | | | |

Longest Common Prefix using Character by Character Matching

# ‹LCP – Força Bruta ›

**04** Step

In the fourth iteration (3rd index) all the characters are not same, so we stop our iteration here and our longest prefix string is "gee".

Result = “gee”

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr[0] = | g | e | e | k | s | f | o | r | g | e | e | k | s |
| arr[1] = | g | e | e | k | s | | | | | | | | |
| arr[2] = | g | e | e | k | | | | | | | | | |
| arr[3] = | g | e | e | z | e | r | | | | | | | |

Longest Common Prefix using Character by Character Matching

# ‹LCP – Busca Binária›

**A ideia é usar busca binaria nos indices para achar o comprimento do maior prefix comum.**
**Podemos observar que esse compimento nunca sera maior que o comprimento da menor string do comjunto. Então a busca sera do indice 0 ate o comprimento da menos string.**

**Em cada iteração da busca [low to high] iremos checar se o prefix de [low to mid] é comum entre todas as strings ou não**

- **Se o prefixo é comum entre todas as string, então procuramos por prefixos de maior comprimento atualizando *low = mid + 1*.**
- ***Senão verifica prefixos menores, atualizando high = mid – 1*.**

**Complexidade de Tempo: O(N*M), onde N é o numero de strings e M é o comprimento da Menor string.**

# ‹LCP – Busca Binária›

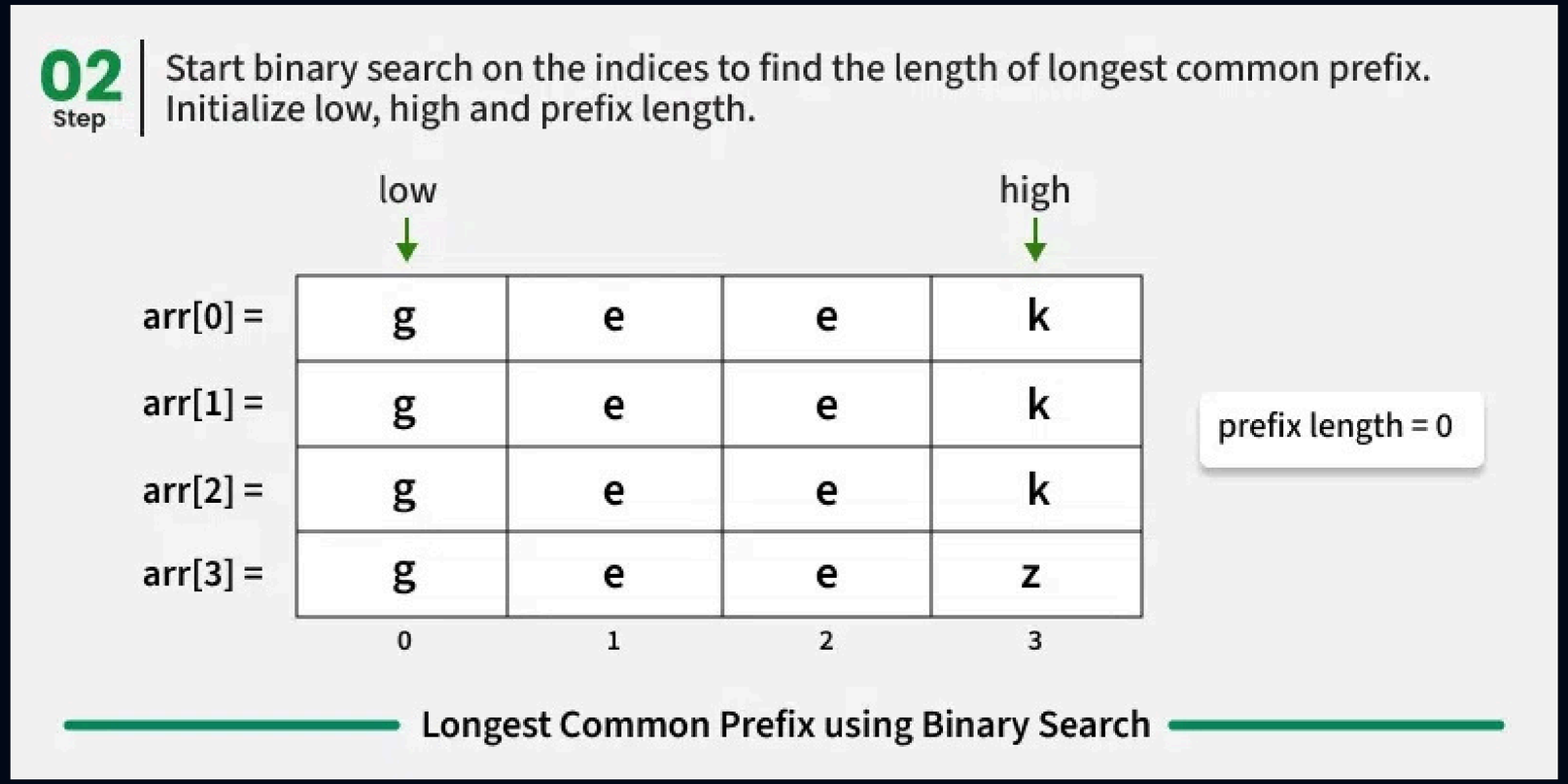
**01** Step

Find the length of shortest string. Since the shortest string "geek" is having length = 4, we will only consider the first 4 characters of all strings.

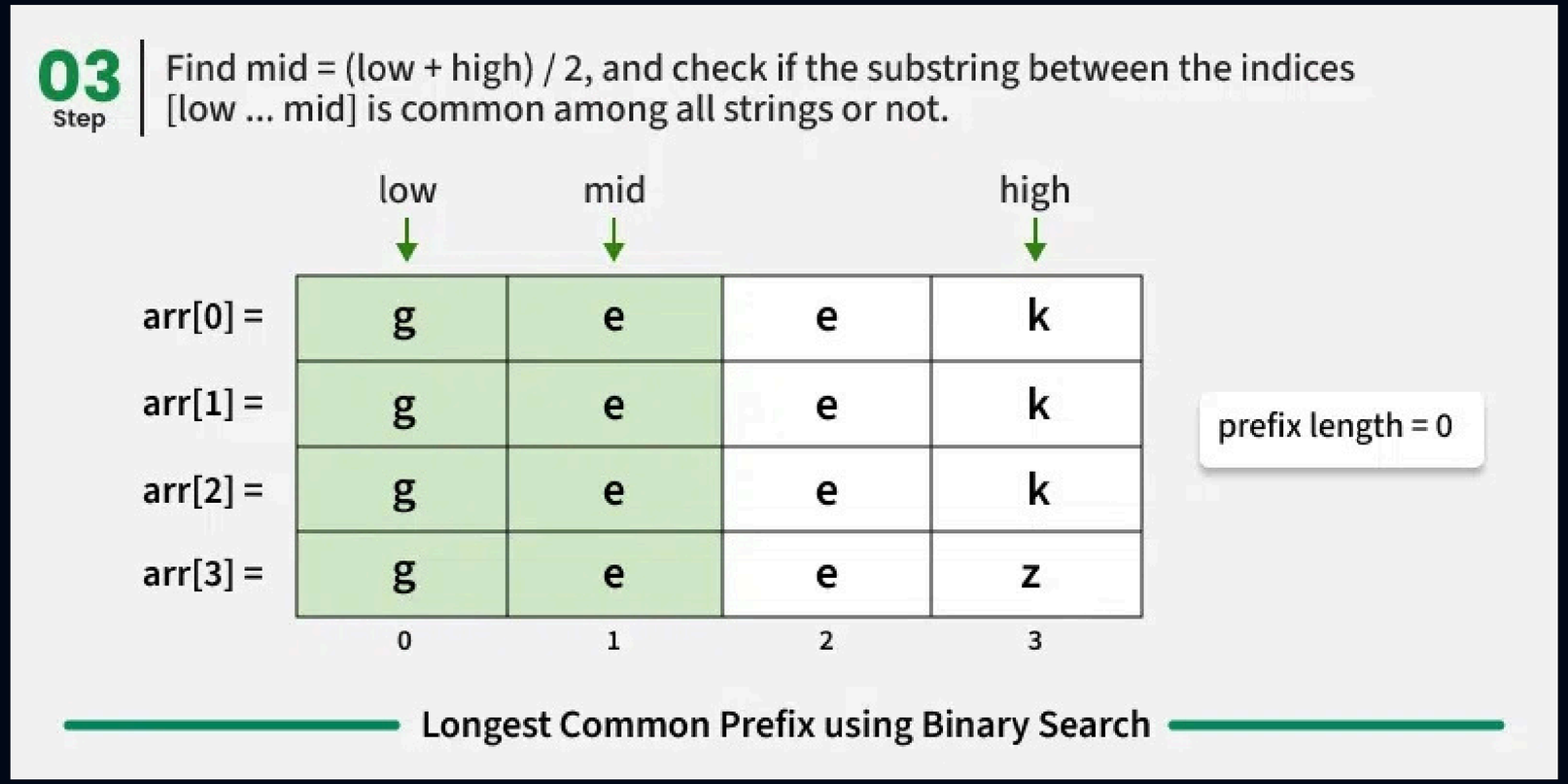
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr[0] = | g | e | e | k | s | f | o | r | g | e | e | k | s |
| arr[1] = | g | e | e | k | s | | | | | | | | |
| arr[2] = | g | e | e | k | | | | | | | | | |
| arr[3] = | g | e | e | z | e | r | | | | | | | |

Longest Common Prefix using Binary Search

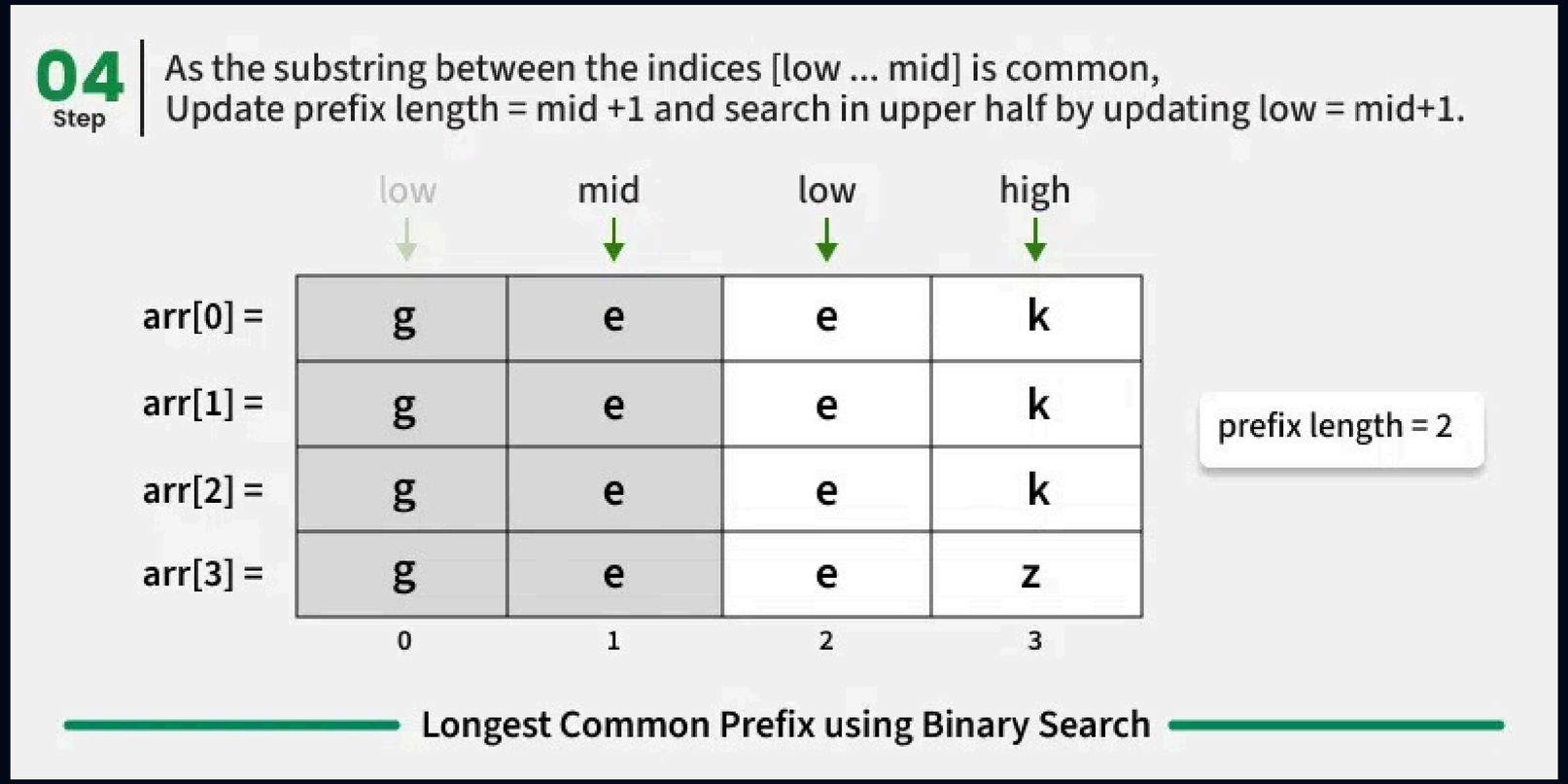
# ‹LCP – Busca Binária›



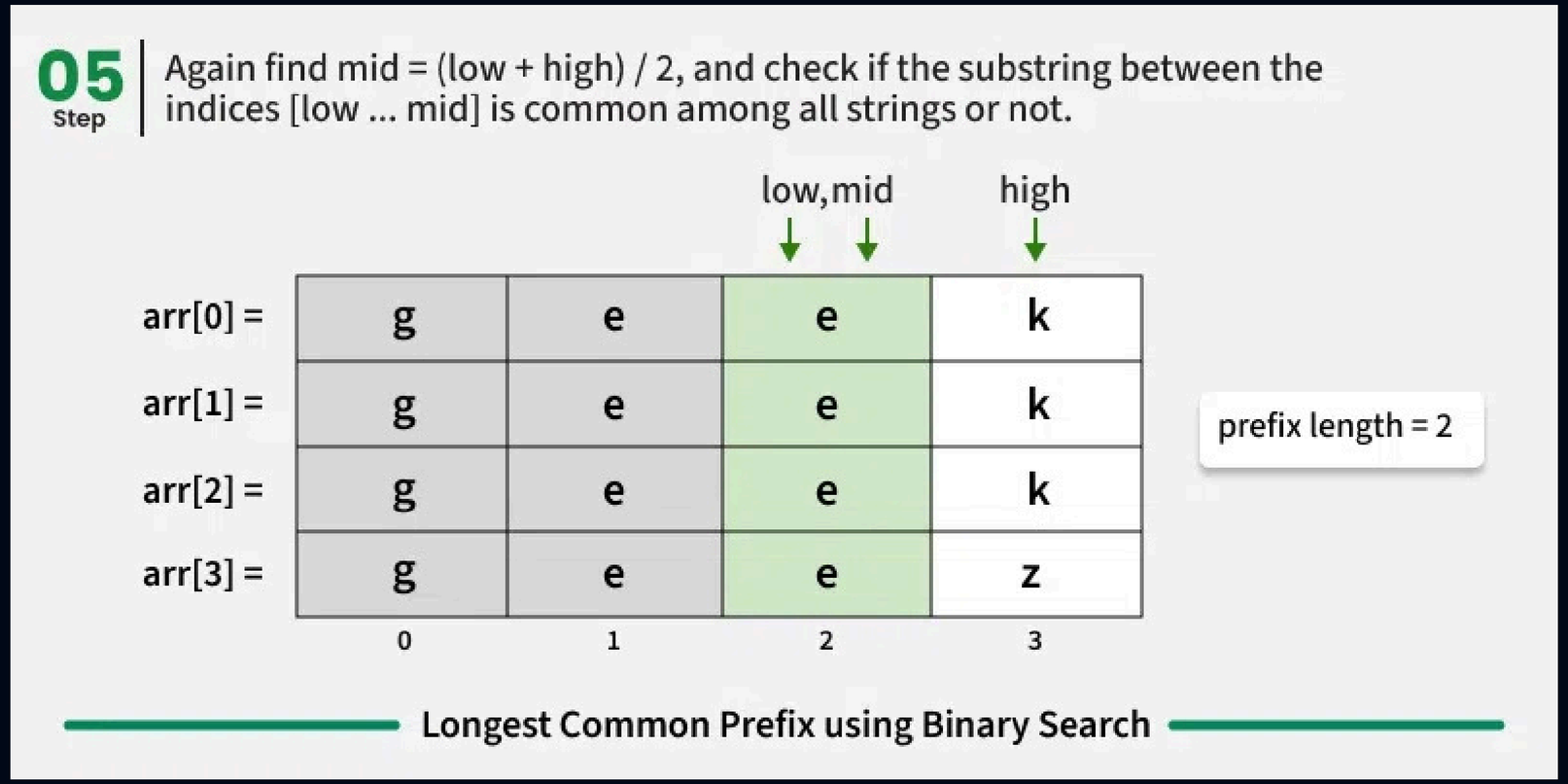
# ‹LCP – Busca Binária›



# ‹LCP – Busca Binária›



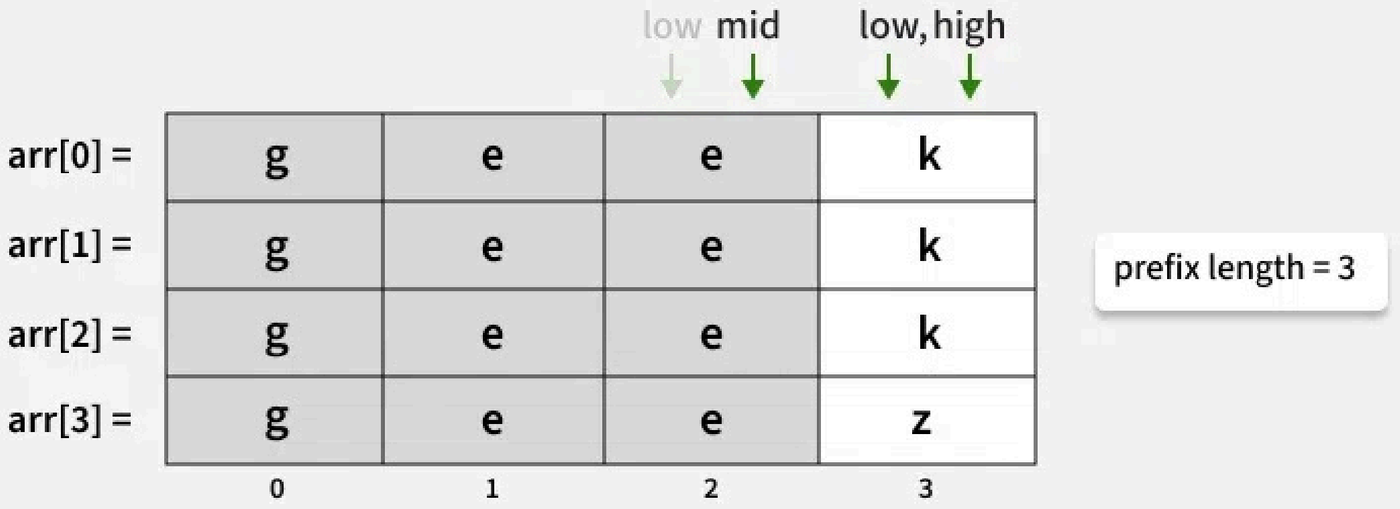
# ‹LCP – Busca Binária›



# ‹LCP – Busca Binária›

**06** Step

As the substring between the indices [low ... mid] is common,
Update prefix length = mid +1 and search in upper half by updating low = mid + 1.

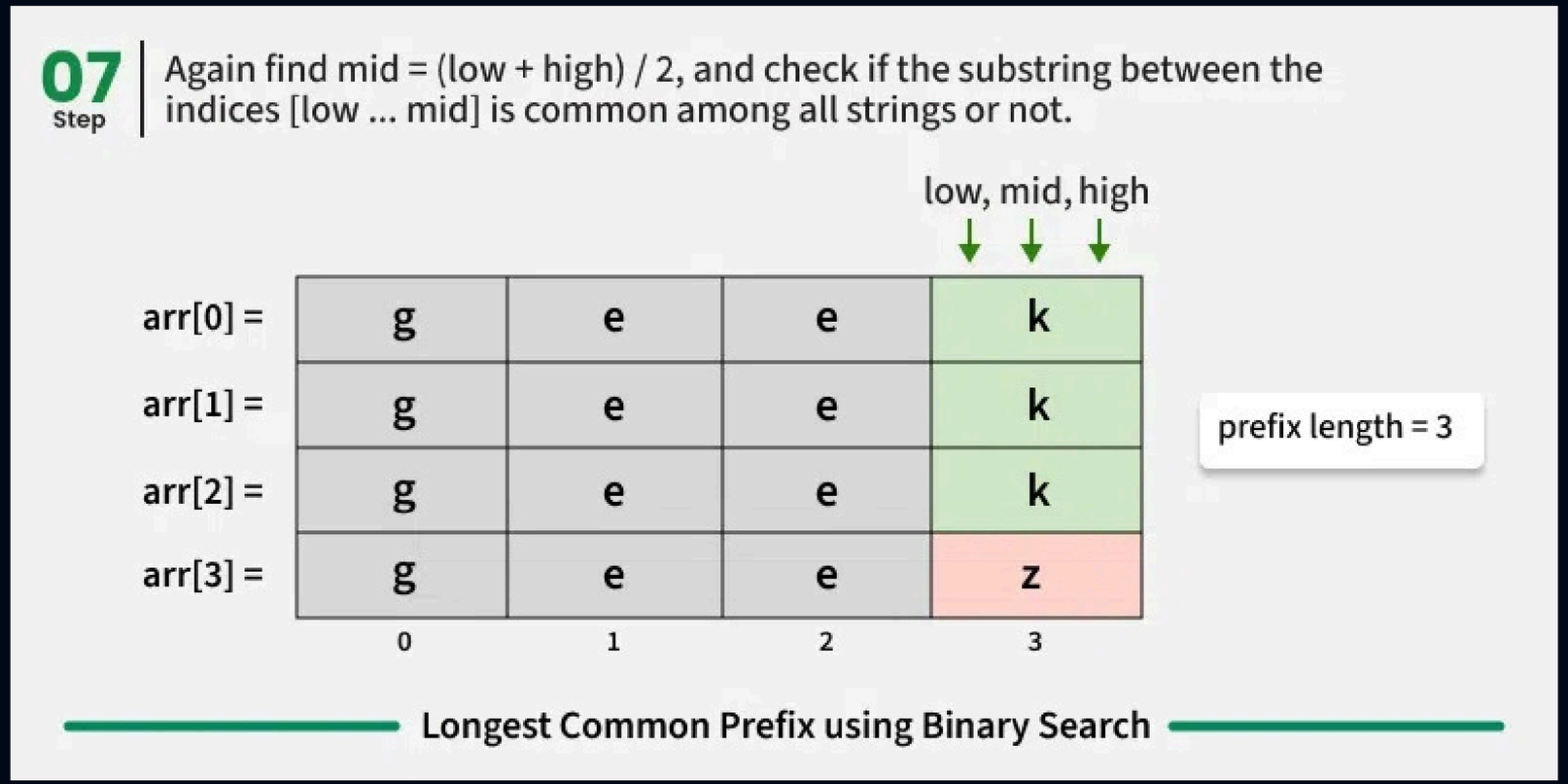



Longest Common Prefix using Binary Search

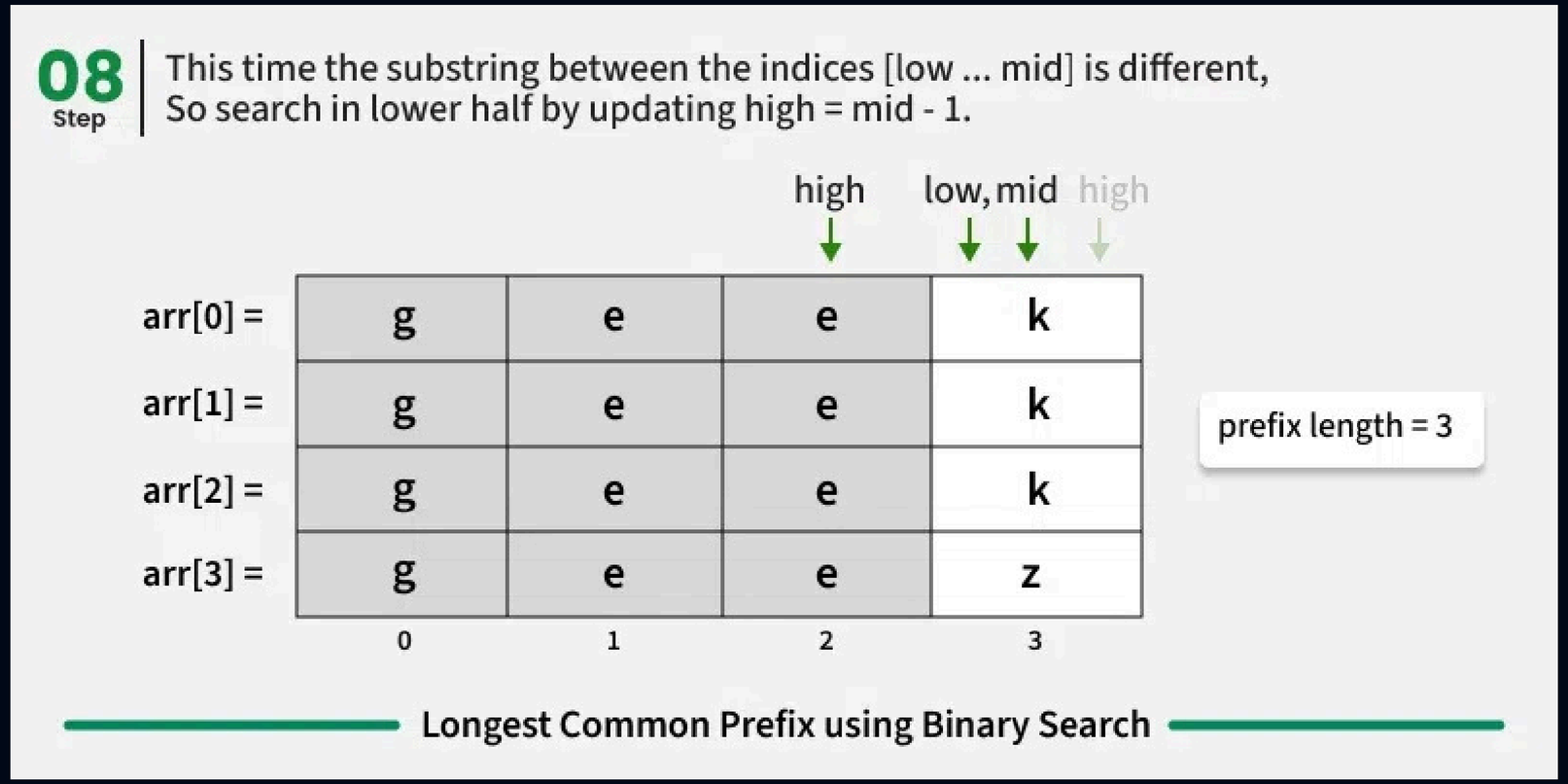
# ‹LCP – Busca Binária›

**07** Step

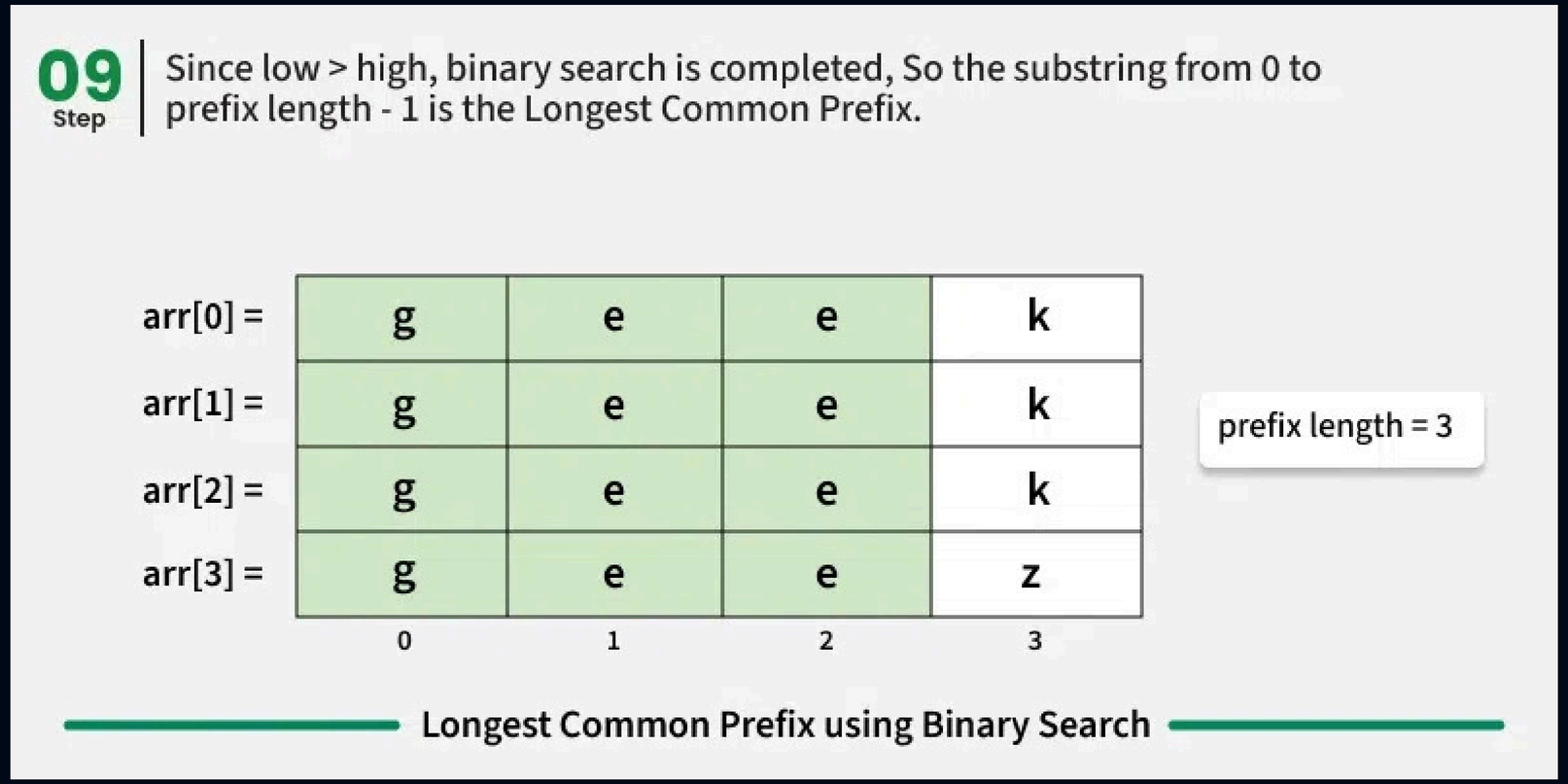
Again find mid = (low + high) / 2, and check if the substring between the indices [low ... mid] is common among all strings or not.



# ‹LCP – Busca Binária›



# ‹LCP – Busca Binária›



# ‹LCP array›

# ‹LCP Array›

LCP Array é um array de tamanho n (como Suffix Array). Um valor lcp[i] indica o comprimento do maior prefixo comum dos sulfixos indexados por sulffix[i] e sulffix[i+1]. suffix[n-1] não é definido por nao possuir elemento depois dele.

# ‹LCP Array›

```
0 banana                         5 a
1 anana  Sort the Suffixes       3 ana
2 nana   ---------------->       1 anana
3 ana     alphabetically         0 banana
4 na                             4 na
5 a                              2 nana
```

```
txt[0..n-1] = "banana"
suffix[]  = {5, 3, 1, 0, 4, 2|
lcp[]     = {1, 3, 0, 0, 2, 0}
Suffixes represented by suffix array in order are:
{"a", "ana", "anana", "banana", "na", "nana"}
lcp[0] = Longest Common Prefix of "a" and "ana"     = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

# Resolução do Problema Motivador

1237 - Comparação de Substring

**A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução!** 😊

# Lista de Exercícios

1237 - Comparação de Substring
2087 - Conjuntos Bons e Ruins


Se tiver alguma dúvida ou dificuldade na resolução de algum exercício, sinta-se à vontade para perguntar! 😊

# Referências

[1] GEEKSFORGEEKS. Prefix Function and KMP Algorithm for Competitive Programming. GeeksforGeeks, [s. l.], [s. d.]. Disponível em: https://www.geeksforgeeks.org/prefix-function-and-kmp-algorithm-for-competitive-programming/. Acesso em: 25 maio 2025.

[2] BRUNO MONTEIRO. Algoritmo de KMP[vídeo]. YouTube, 29 jul. 2022. Disponível em: https://www.youtube.com/watch?v=RXISWaGmYW8. Acesso em: 25 maio 2025