CodeIT

Lesson 5 Objectives:

To gain an understanding of:

1. What the **substring search / match** problem is

2. How famous algorithms like **Knuth Morris Pratt & Boyer-Moore** are used to solve this problem

Substring Search

Substring Search

Given a **text** and a **substring**, find all instances of the substring in the text

Substring Search

Given a **text** and a **substring**, find all instances of the substring in the text

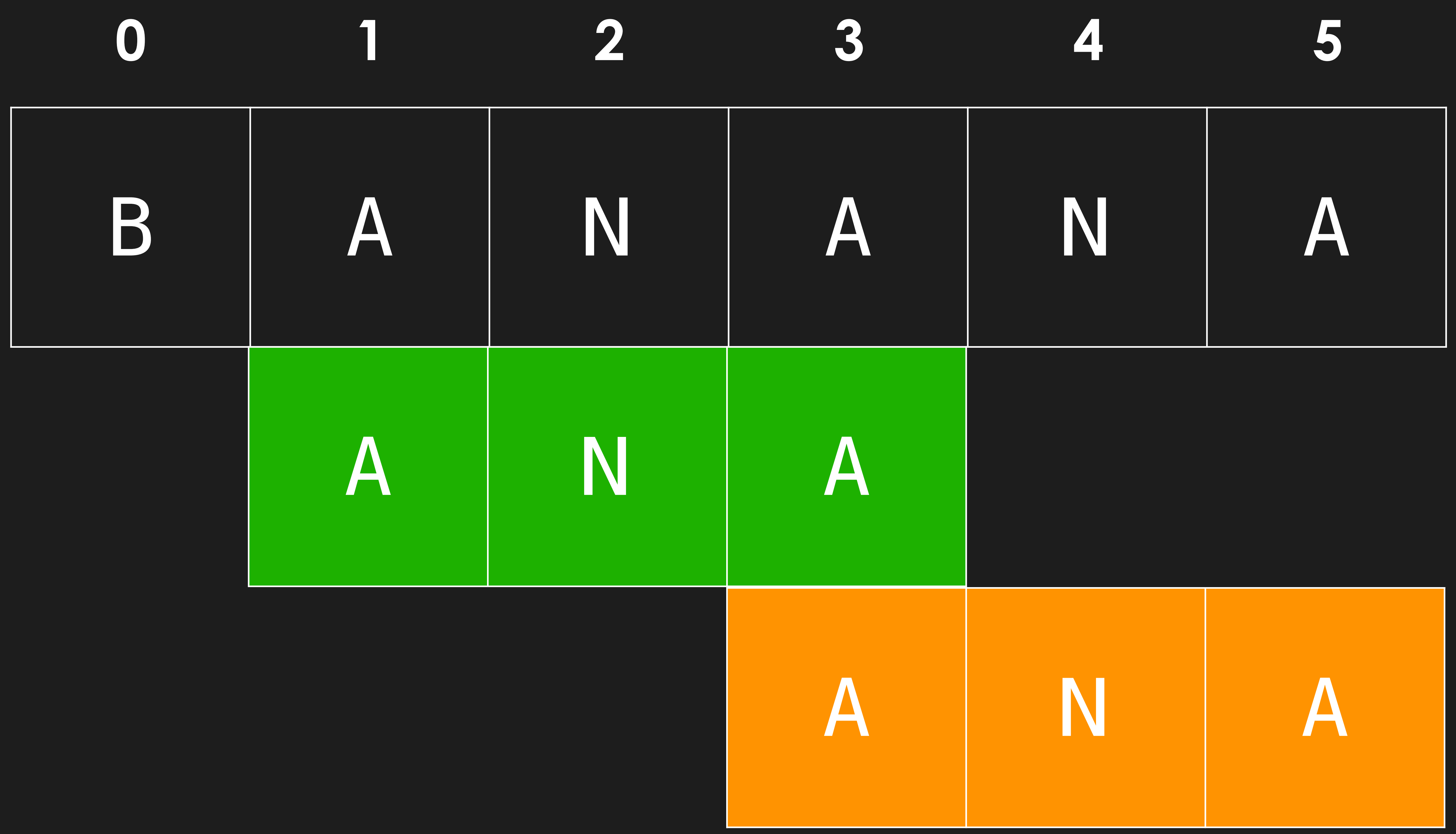
Example

text: "BANANA"

substring: "ANA"

0	1	2	3	4	5
B	A	N	A	N	A

0	1	2	3	4	5
B	A	N	A	N	A
	A	N	A		


0 1 2 3 4 5
B A N A N A
A N A
A N A

Result: Index **1** & **3**

Why Substring Search?

Why Substring Search?

Among the many usages of substring search (Text Editor Search, etc.), **gene sequence pattern finding** is a good example to illustrate the usefulness of such an algorithm

Why Substring Search?

Imagine if we had a **gene sequence** and we wanted to find all instances of a particular **pattern of chromosomes** in that sequence:

sequence: `"TTTGTTTAGGGTACCATCAGGA"`
substring: `"GTAC"`

Why Substring Search?

Imagine if we had a **gene sequence** and we wanted to find all instances of a particular **pattern of chromosomes** in that sequence:

sequence: `"TTTGTTTAGGGTACCATCAGGA"`
substring: `"GTAC"`

It is important to have not just an algorithm which solves this problem, but an efficient one, because gene sequences can be extremely long!

Algorithm 0: Brute Force Approach

Algorithm 0: Brute Force Approach

Imagine we have a **text** with **N chars** and a **substring** with **K chars**

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

CodeIT

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

CodeIT

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. **For each character, check if:** `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. **For each character, check if:** `text[i : i + k] == substring`

substring: `"ADAG"`

	i													
A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

	A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

	i													
A	D	C	A	A	D	A	G	I	E	A	D	A	F	H
	A	D	A	G										

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

		i												
A	D	C	A	A	D	A	G	I	E	A	D	A	F	H
		A	D	A	G									

Brute Force Approach

1. Iterate through each character of the text
2. **For each character, check if:** `text[i : i + k] == substring`

substring: `"ADAG"`

		i												
A	D	C	A	A	D	A	G	I	E	A	D	A	F	H
		A	D	A	G									

Brute Force Approach

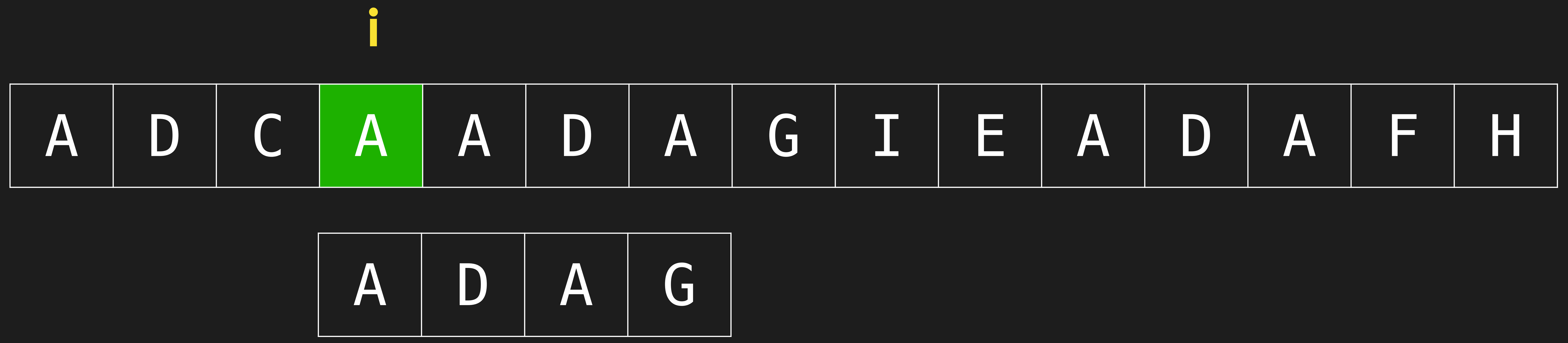
1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

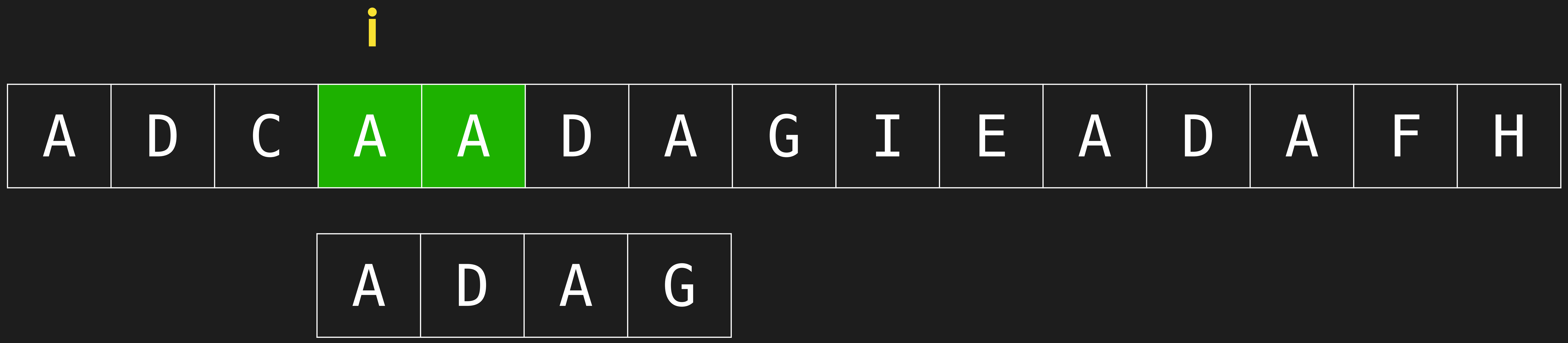
substring: `"ADAG"`



Brute Force Approach

1. Iterate through each character of the text
2. **For each character, check if:** `text[i : i + k] == substring`

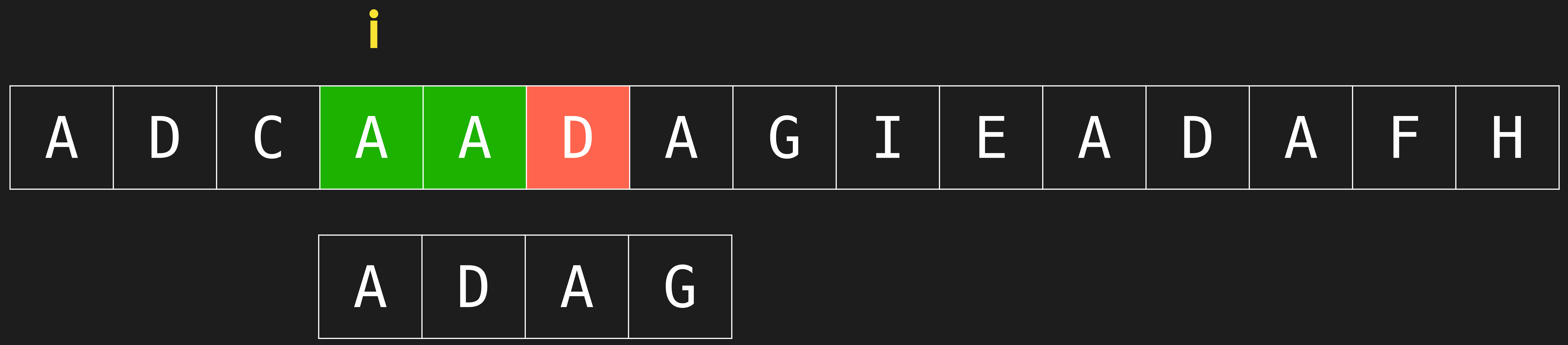
substring: `"ADAG"`



Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`



Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: "ADAG"

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

				i										
A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

i

A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

Brute Force Approach

1. Iterate through each character of the text
2. For each character, check if: `text[i : i + k] == substring`

substring: `"ADAG"`

				i										
A	D	C	A	A	D	A	G	I	E	A	D	A	F	H

A	D	A	G

substring found at i!

Implementation of Brute Force Approach

Brute Force

```python
def BruteForceSubstringSearch(text, substring):
    N = len(text)
    M = len(substring)

    for i in range(N - M + 1):

        match = True
        for j in range(len(substring)):
            if text[i + j] != substring[j]:
                match = False

        if match:
            print("substring found at index {}".format(i))
```

Analysis of Brute Force Approach

Analysis of Brute Force Approach

Best case: Every letter in text is different from substring except last k letters

Analysis of Brute Force Approach

Best case: Every letter in text is different from substring except last k letters

substring: **"AAAA"**

B	B	B	B	B	B	B	B	B	B	B	A	A	A	A

Analysis of Brute Force Approach

Best case: Every letter in text is different from substring except last k letters

substring: **"AAAA"**

B	B	B	B	B	B	B	B	B	B	B	A	A	A	A
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A

One pass through the text: O(N) time!

Analysis of Brute Force Approach

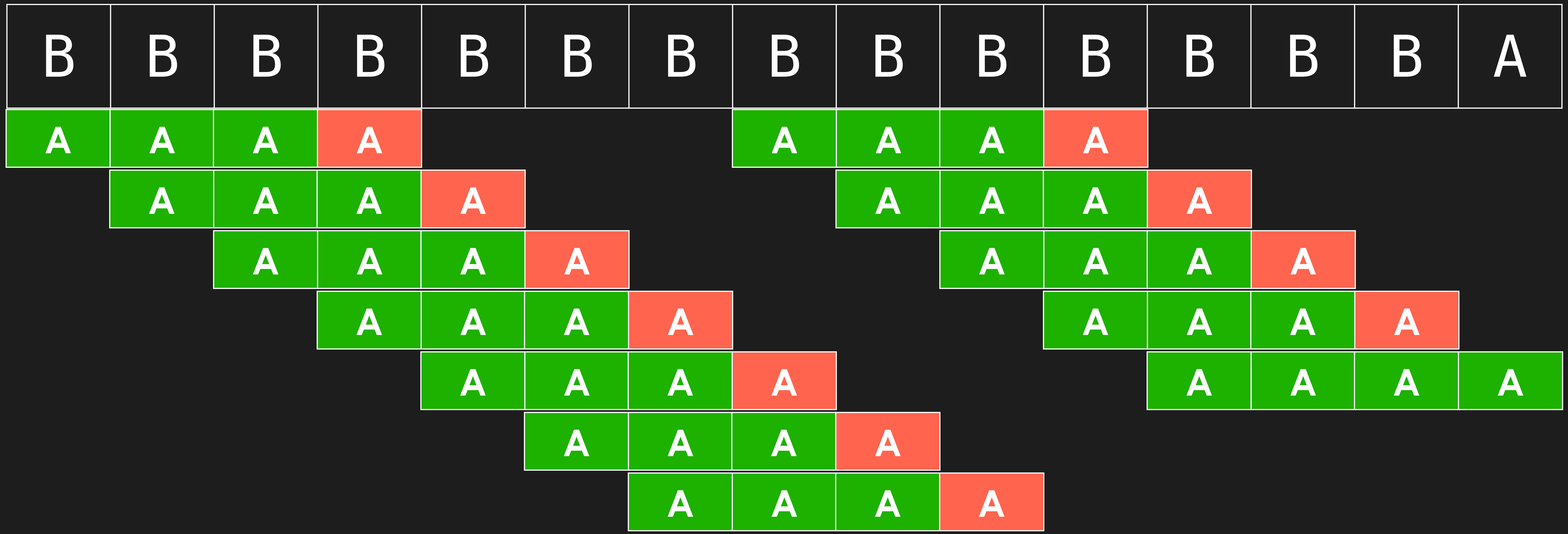
Worst case:

substring: "BBBA"

B	B	B	B	B	B	B	B	B	B	B	A	A	A	A

Analysis of Brute Force Approach

substring: "BBBA"



For each char in text, we check against substring (K - 1 times)
Time complexity: O(NK)

Can we achieve substring search in linear time?

Knuth Morris Pratt Algorithm

Knuth Morris Pratt's algorithm allows us to perform substring search in **linear time** (according to length of text)

On a high level, what KMP does is that every time we encounter a mismatch as we iterate through the **text**, we are aware of how much of the **existing suffix** belongs to the substring pattern

Example:

text

i

A	D	A	D	A	D	A	B	B

substring

j

A	D	A	B	B
0	1	2	3	4

Example:

text

i

A	D	A	D	A	D	A	B	B

substring

j

A	D	A	B	B
0	1	2	3	4

Example:

text

i

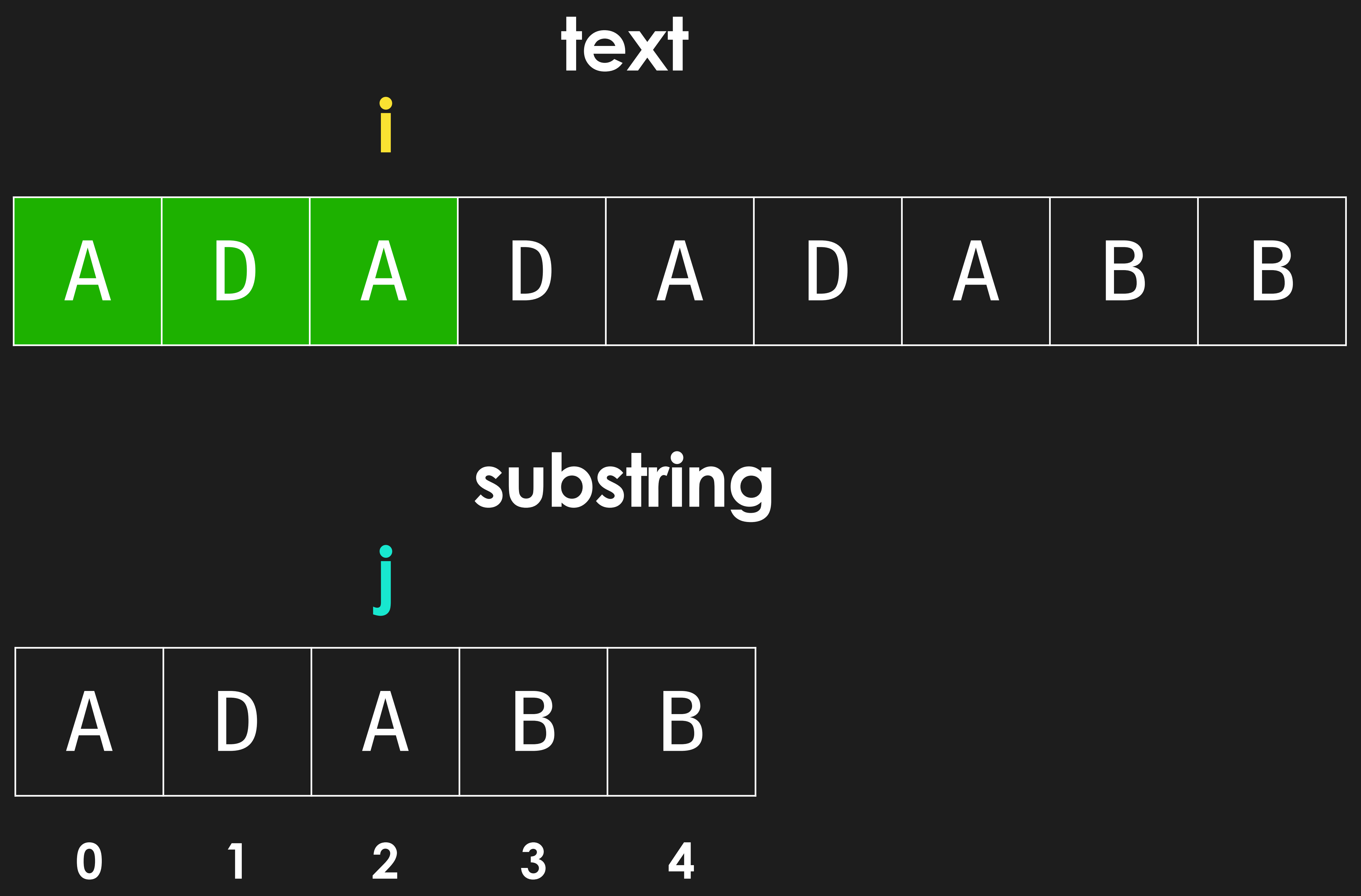
A	D	A	D	A	D	A	B	B

substring

j

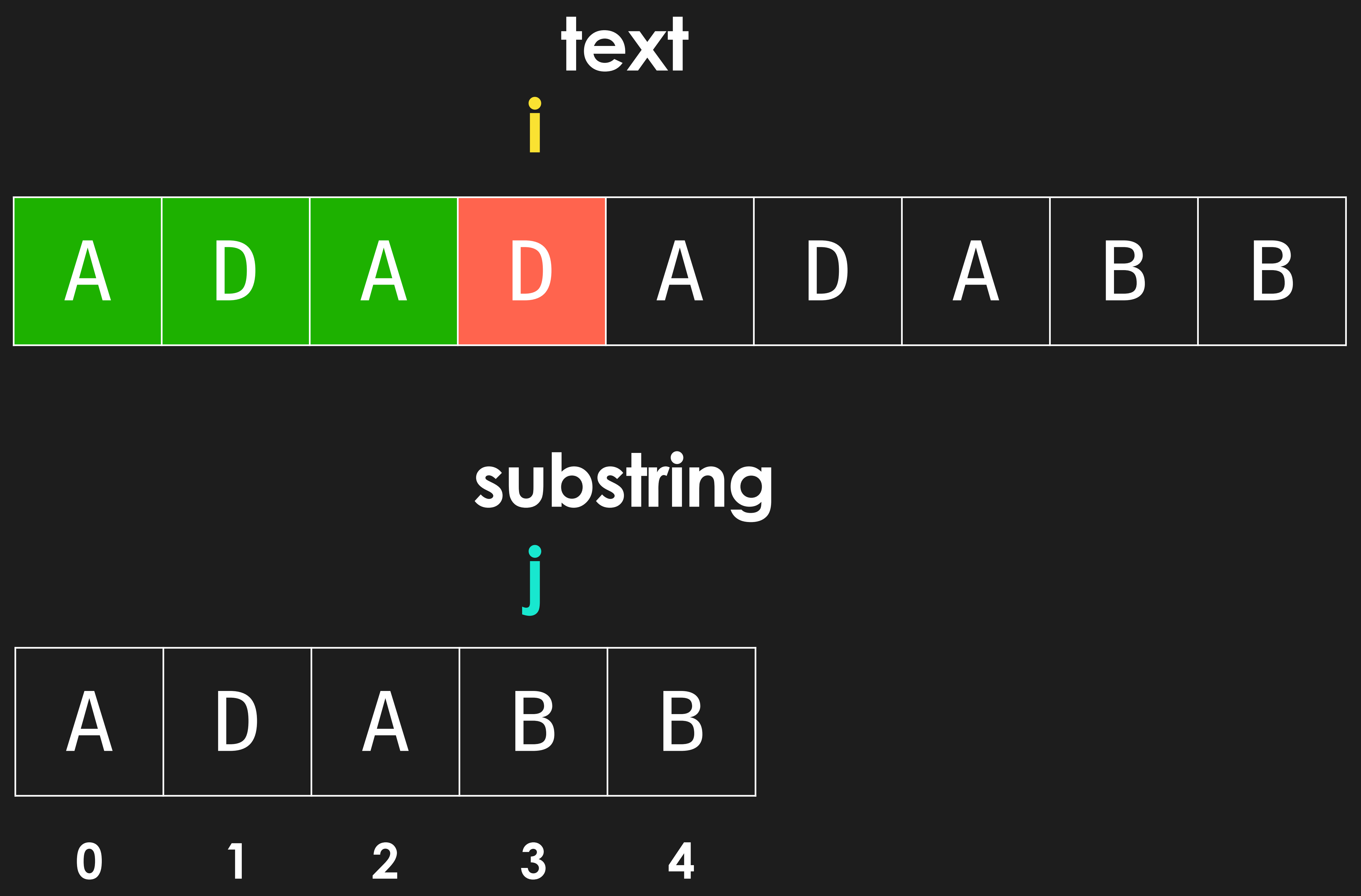
A	D	A	B	B
0	1	2	3	4

Example:



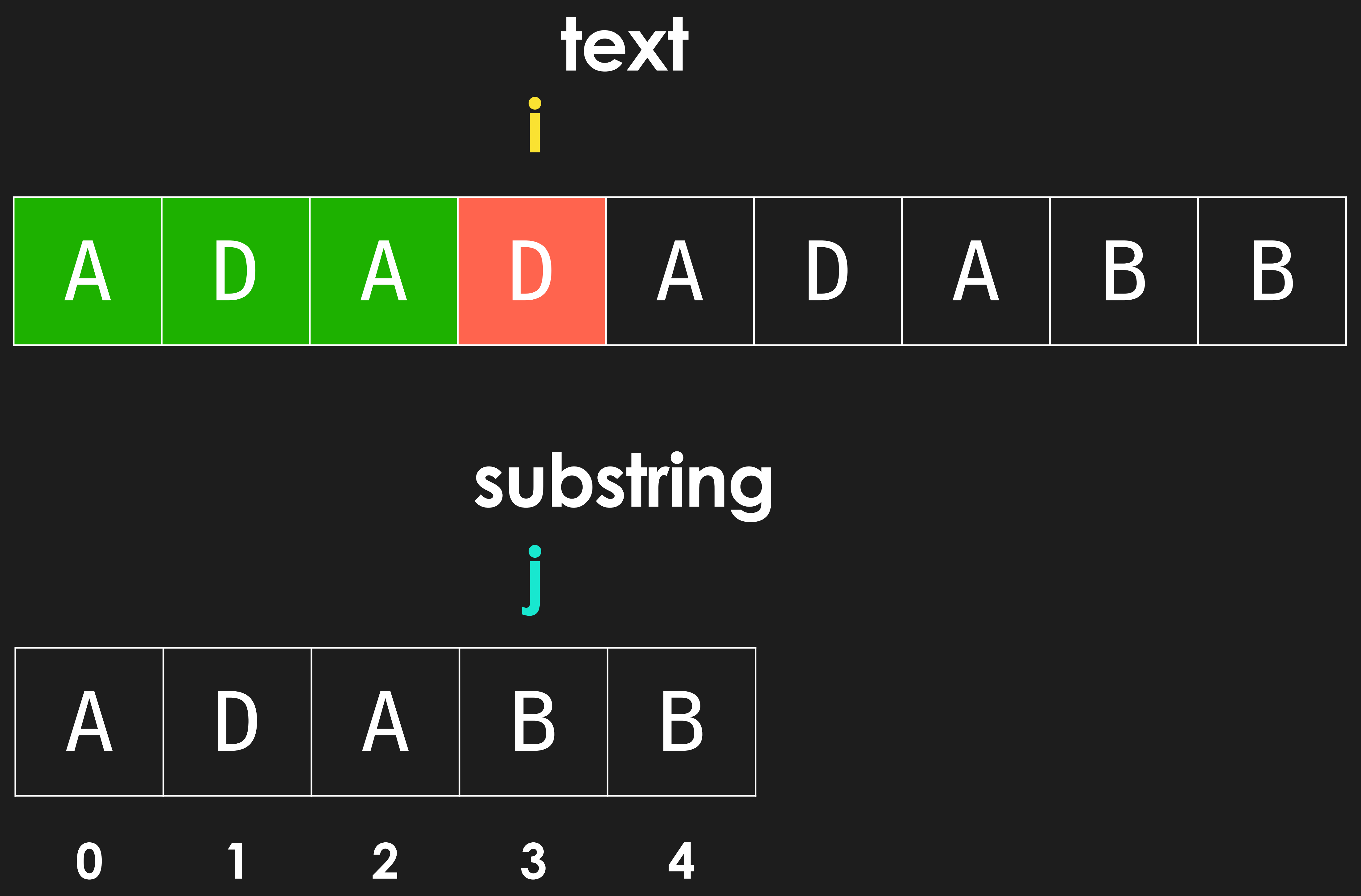
CodeIT

Example:



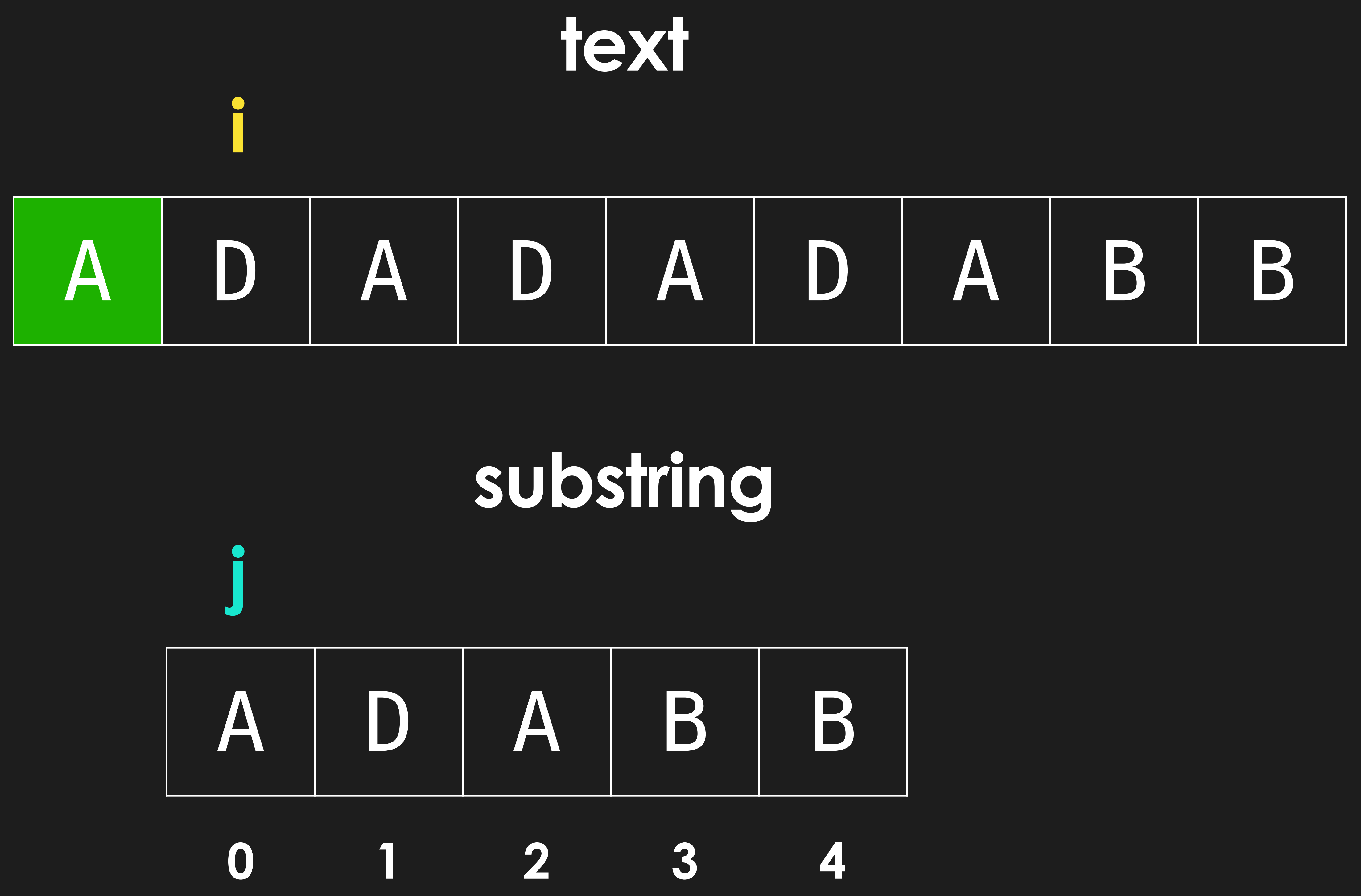
Mismatch!

Example:



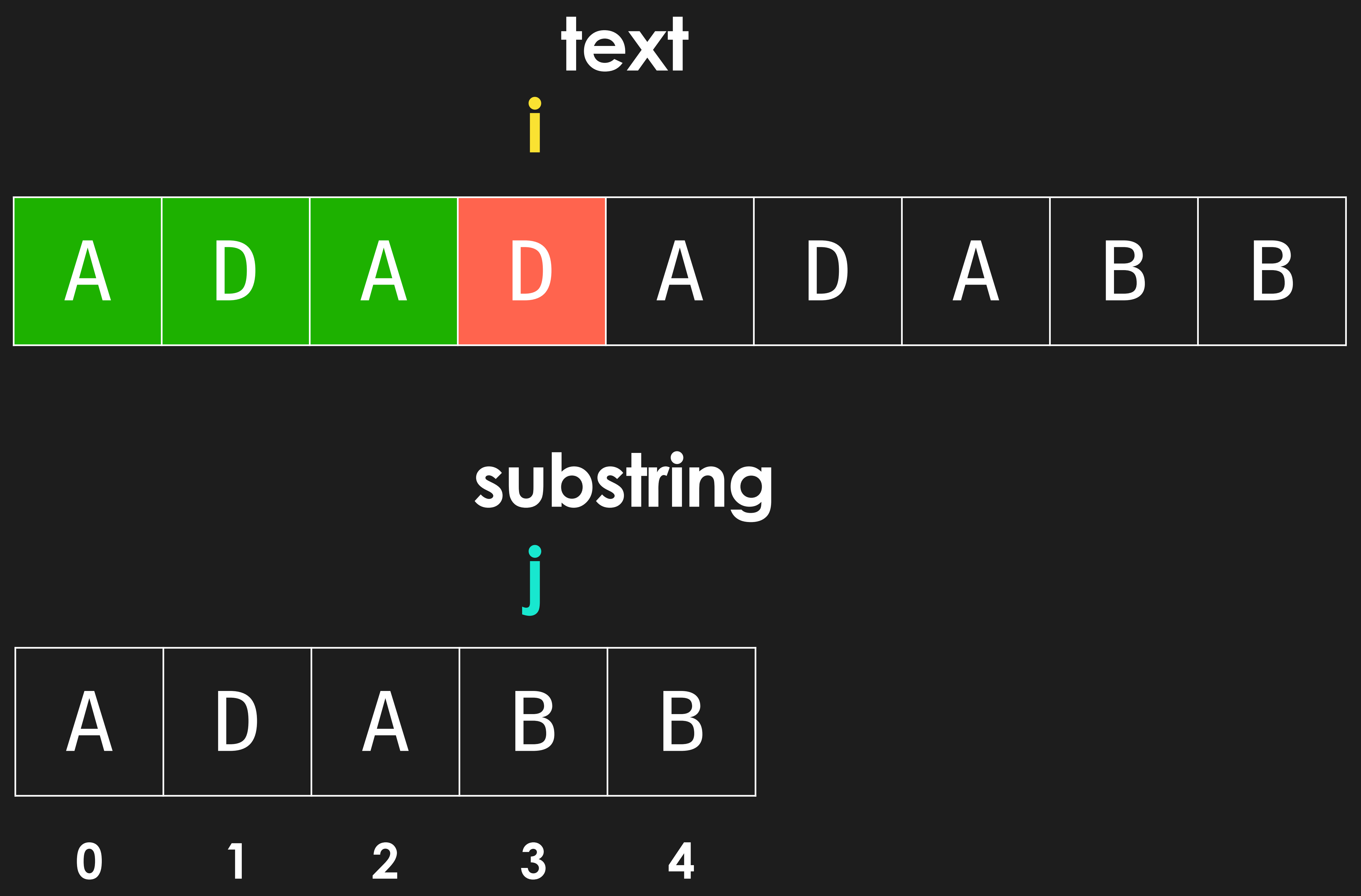
Using brute force, we may do the following

Example:



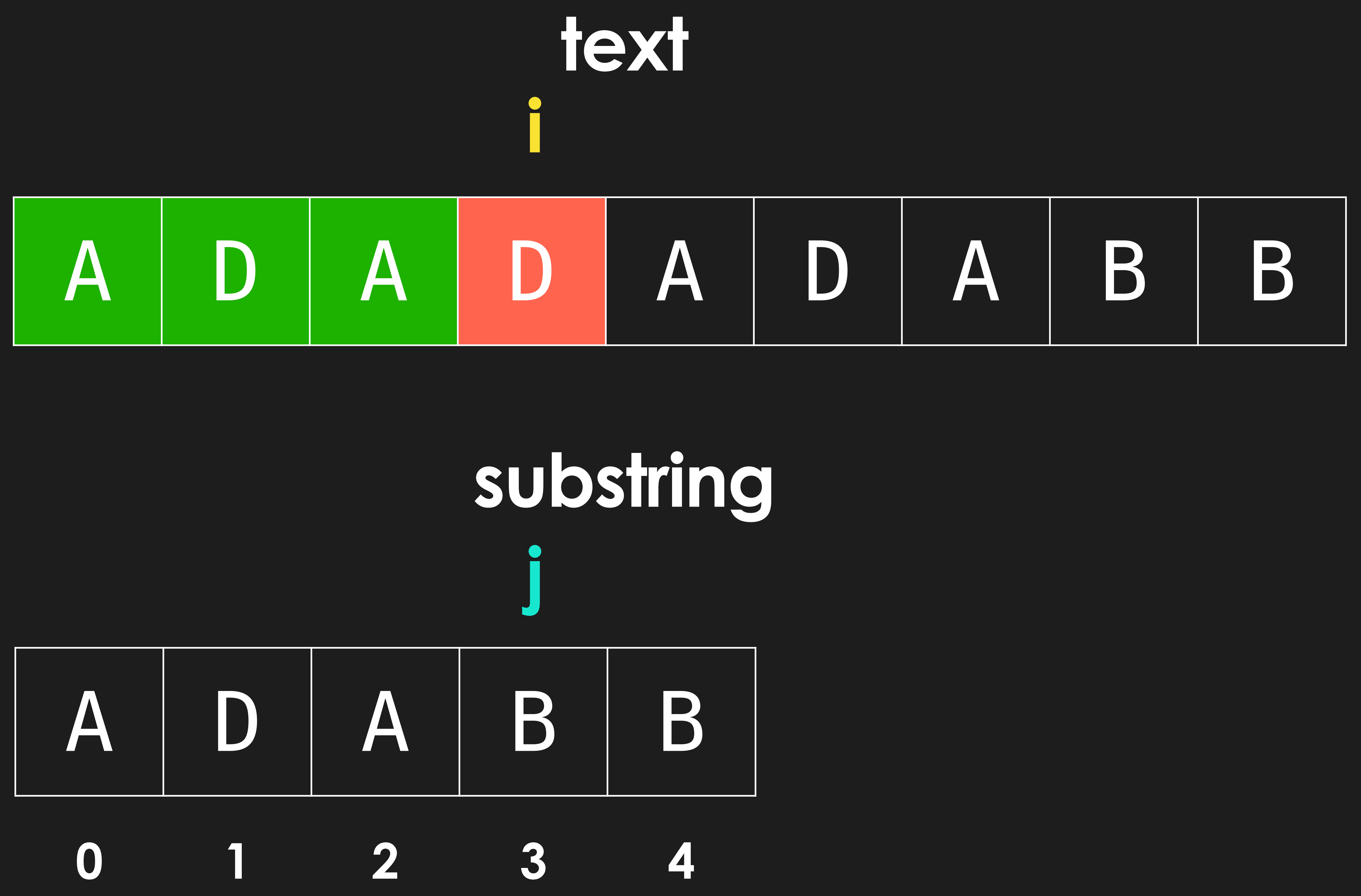
Using brute force, we may do the following

Example:



However, what KMP discovered is that when we have iterated up till text[i], we are aware of all characters from [0 : i - 1]

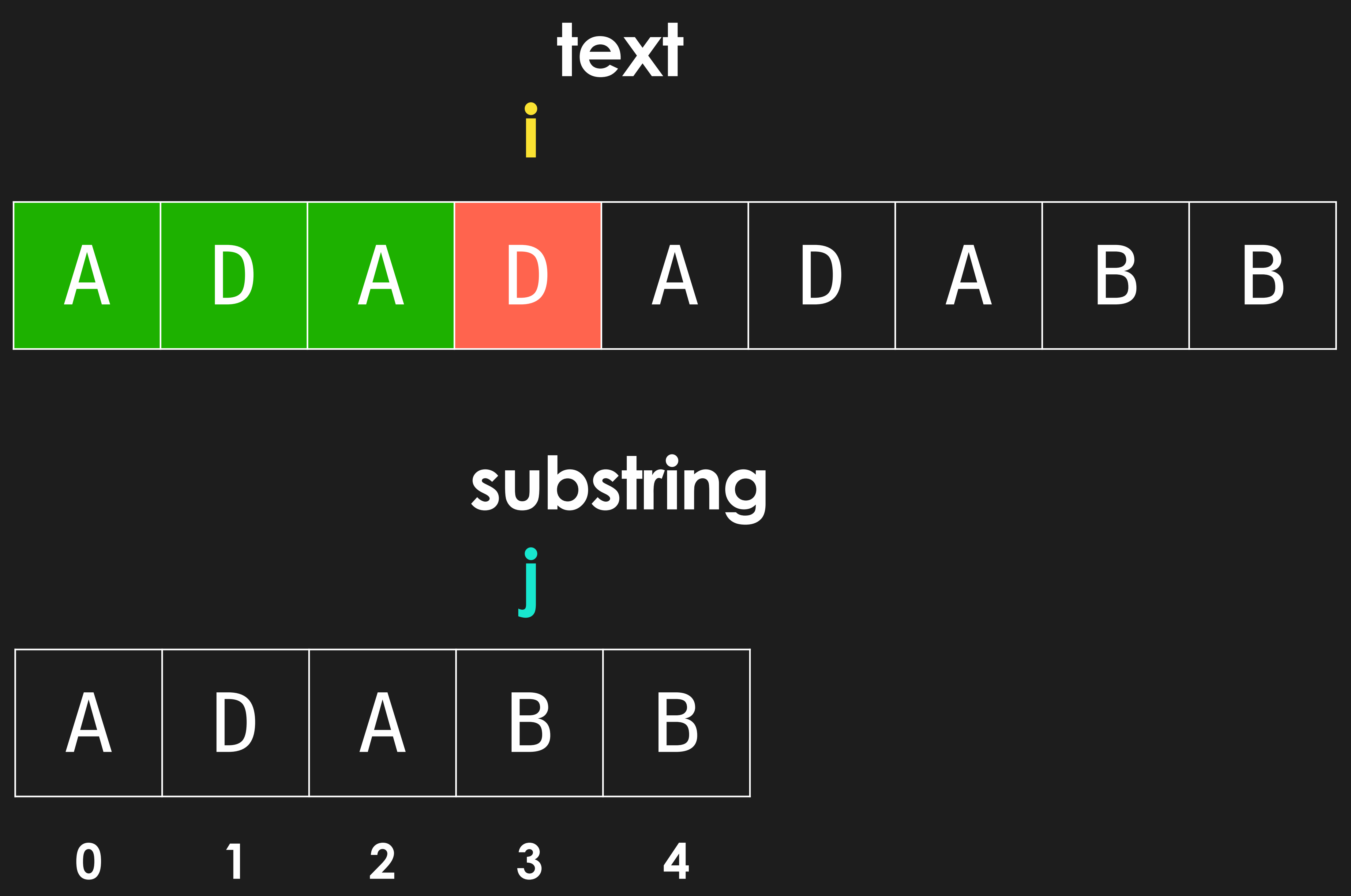
Example:



However, what KMP discovered is that when we have iterate up till text[i], we are aware of all characters from [0 : i - 1]

As such, all we need to do is find the **longest suffix so far** which **matches a prefix of our substring**

Example:



Of all existing **suffixes** in the chars we have iterated over, our longest substring prefix match is **"AD"**

Example:

text

i

A	D	A	D	A	D	A	B	B

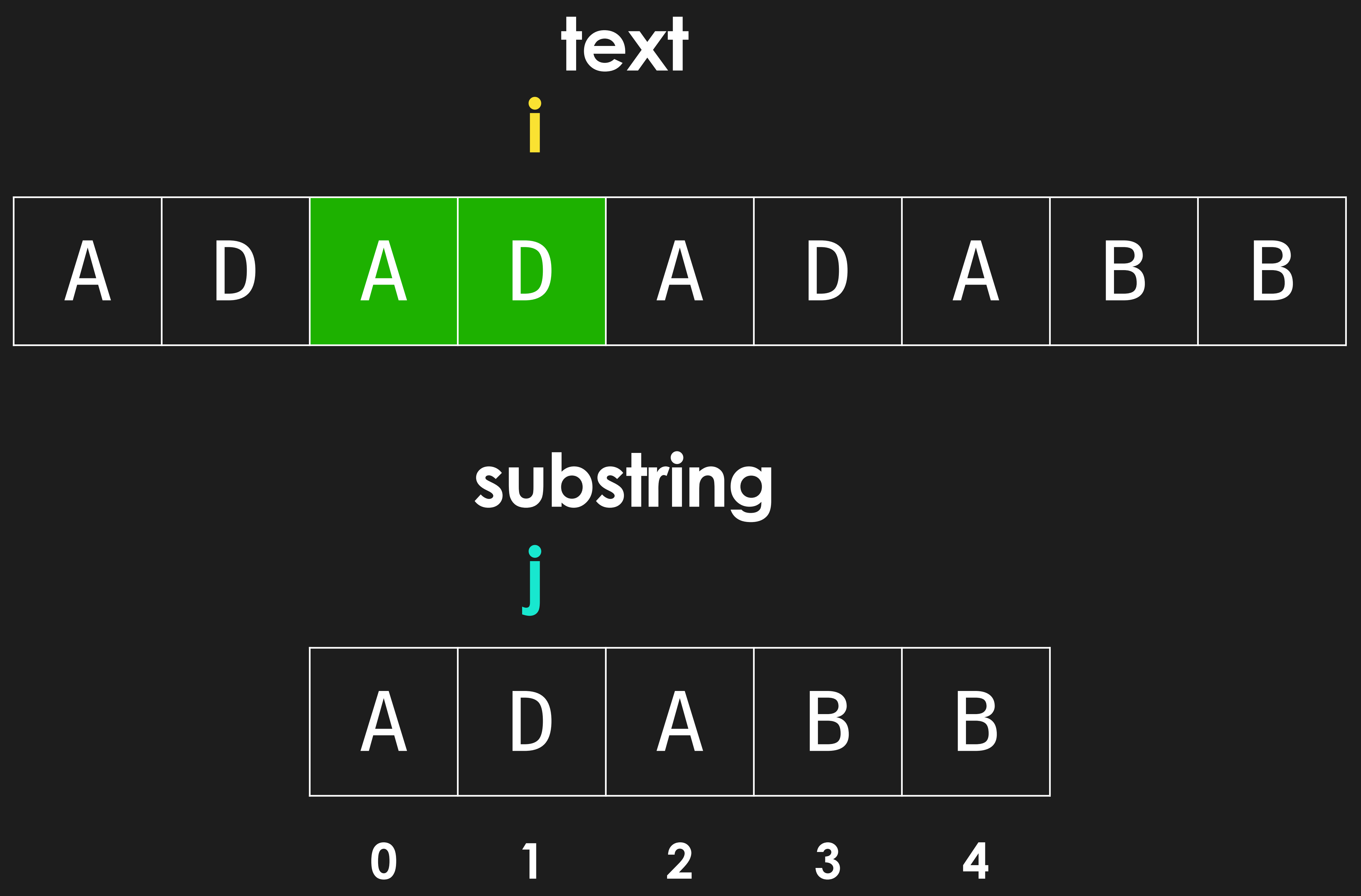
substring

j

A	D	A	B	B
0	1	2	3	4

Of all existing **suffixes** in the chars we have iterated over, our longest substring prefix match is **"AD"**

Example:



Now we can simply continue to increment i, knowing that the **first two chars of the substring have already been matched**

Example:

text

i

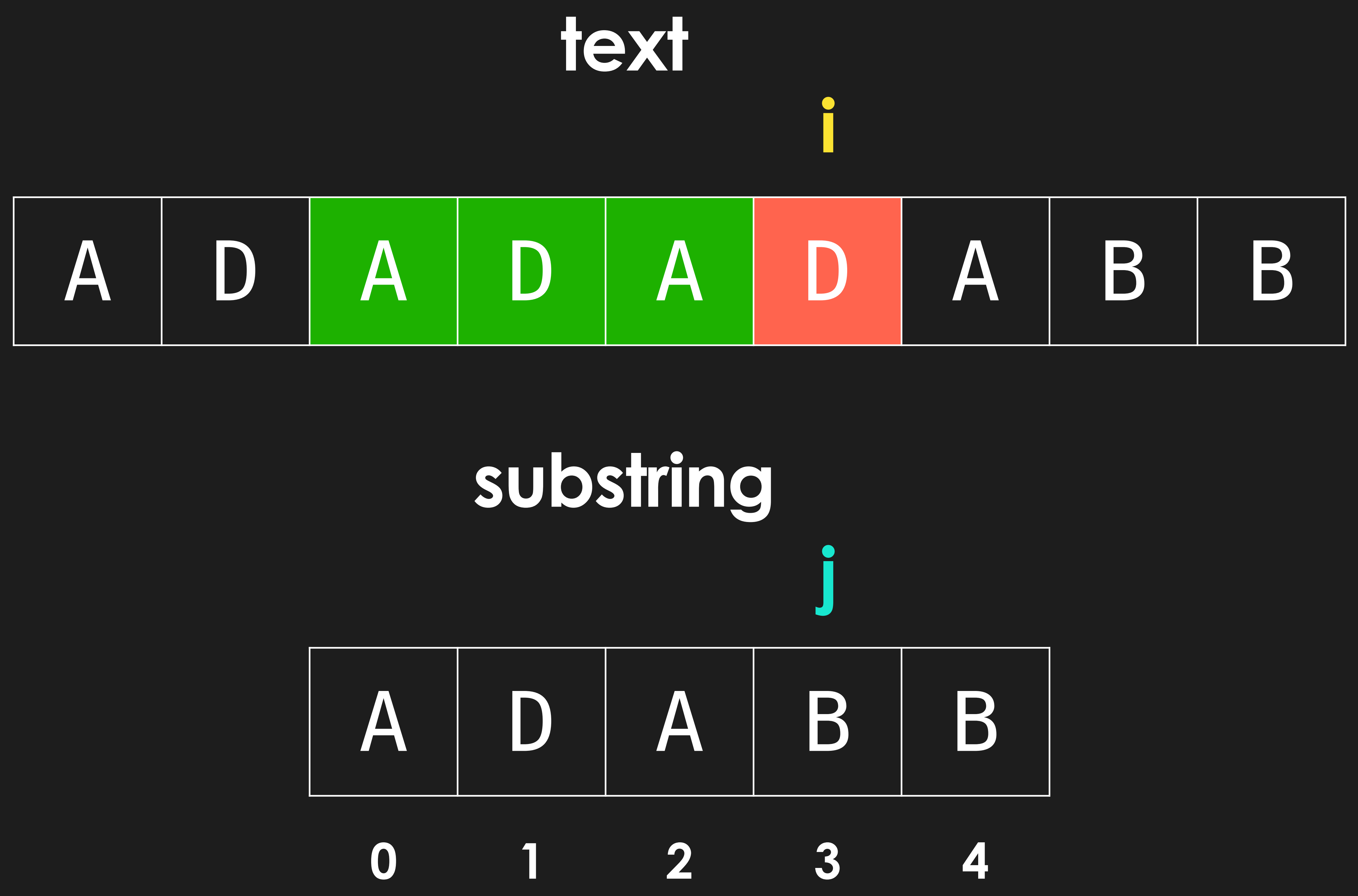
A	D	A	D	A	D	A	B	B

substring

j

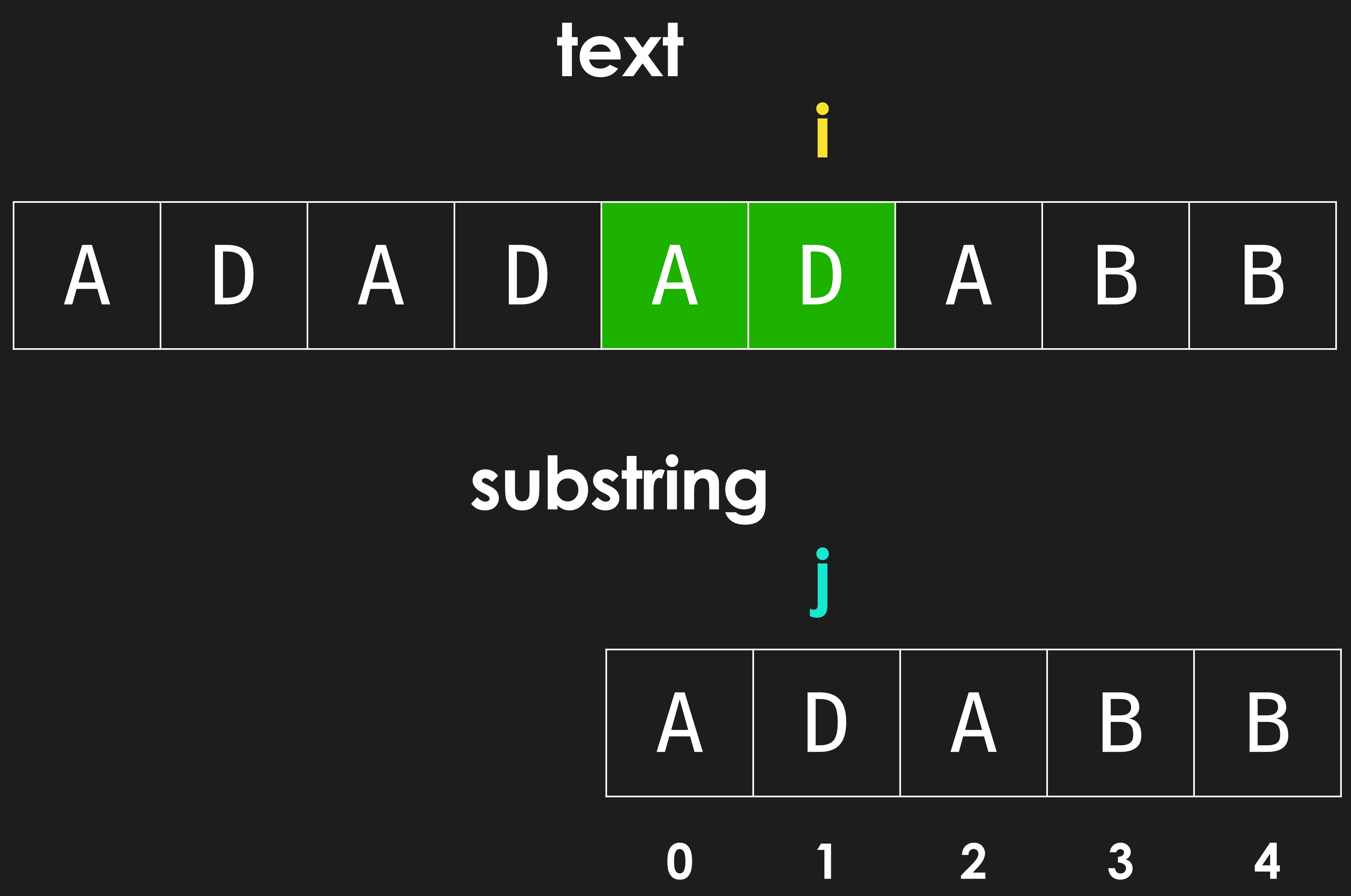
A	D	A	B	B
0	1	2	3	4

Example:



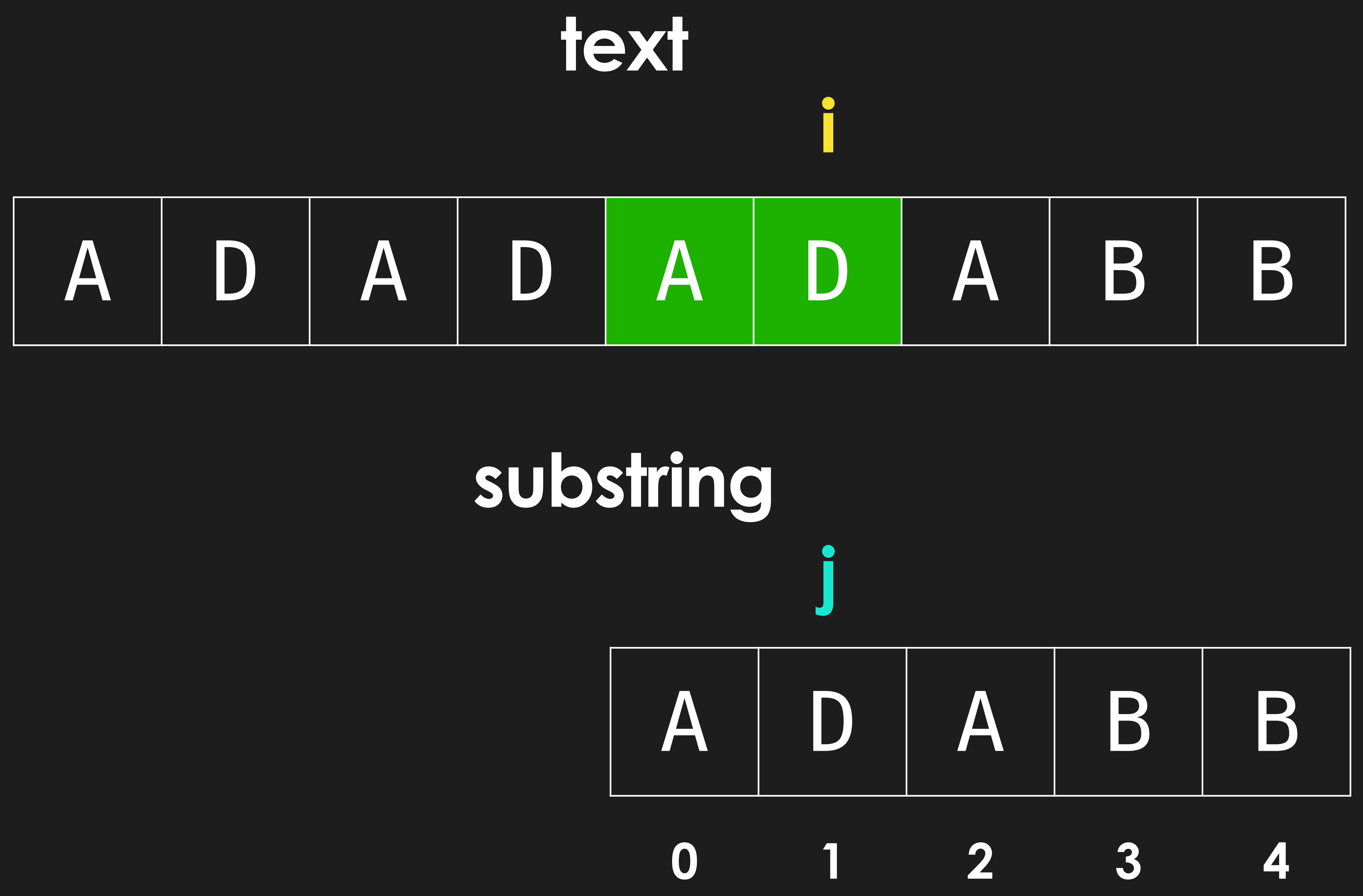
Once again, of all existing **suffixes**, our closest match to the pattern is **"AD"**

Example:



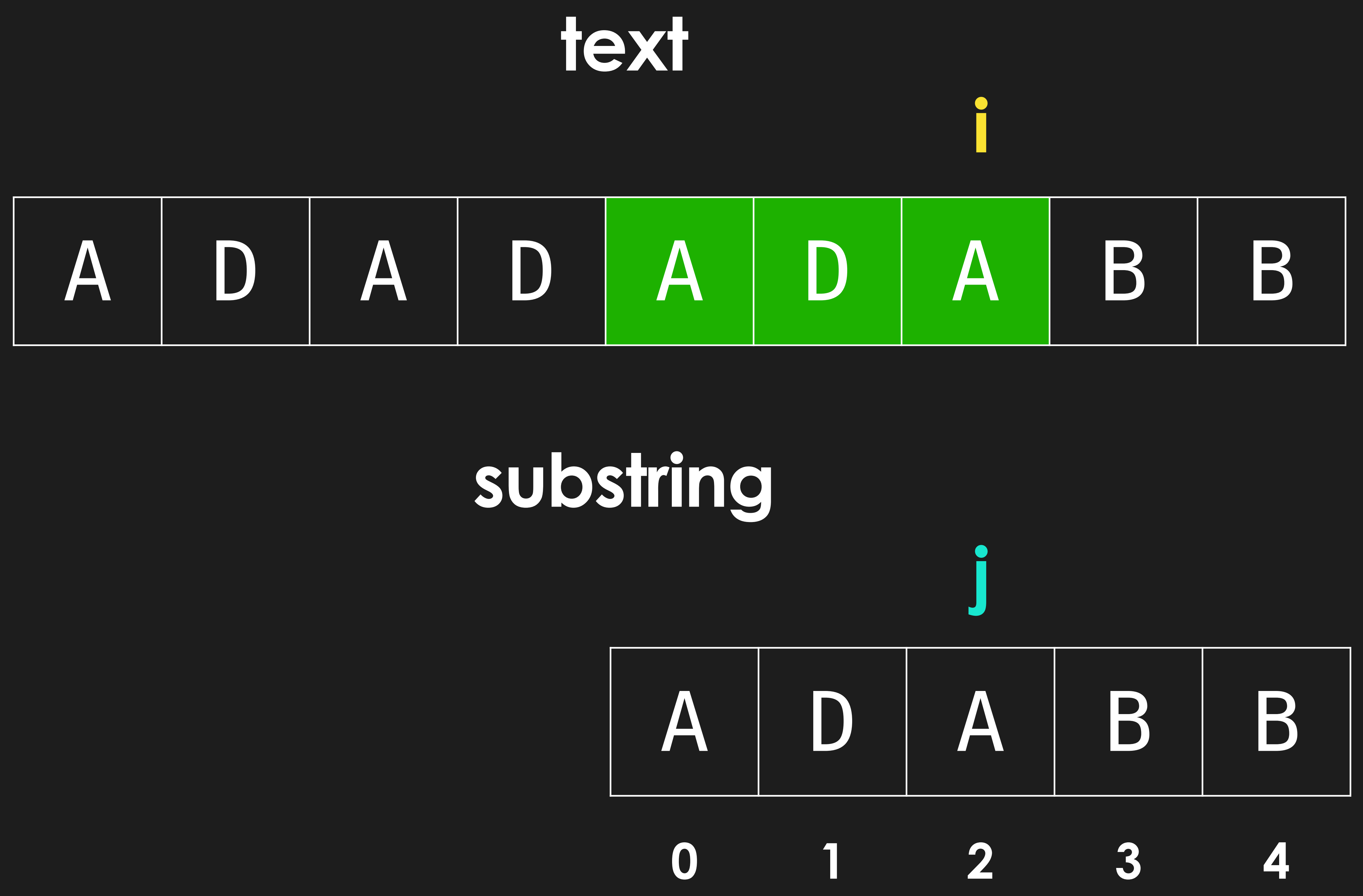
Once again, of all existing **suffixes**, our longest substring prefix match is **"AD"**

Example:



We start pattern matching from index 2 of the substring

Example:



We start pattern matching from index 2 of the substring

Example:

text

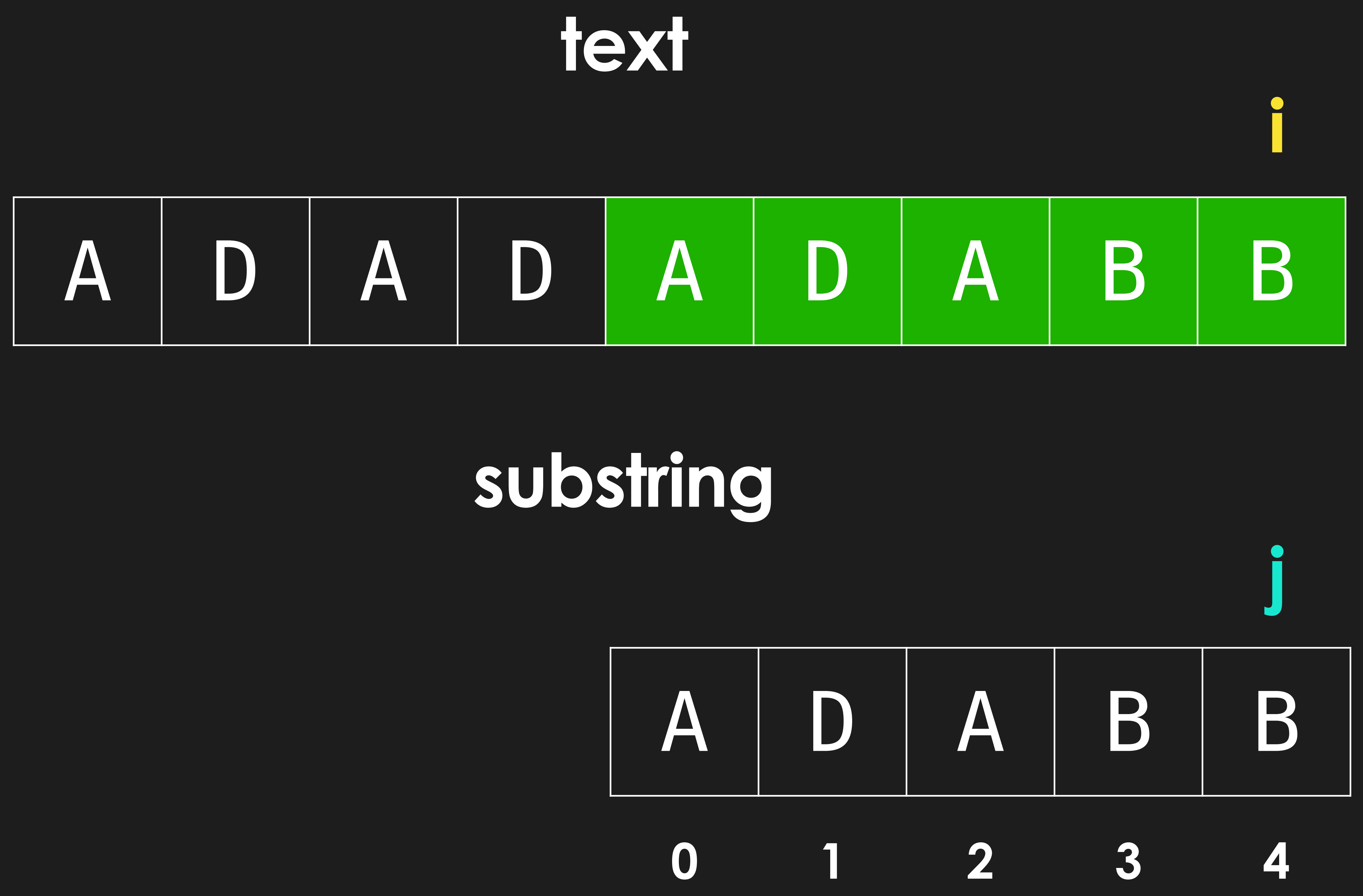
							i	
A	D	A	D	A	D	A	B	B

substring

			j	
A	D	A	B	B
0	1	2	3	4

We start pattern matching from index 2 of the substring

Example:



We have found a substring!

The Problem: How do we know what the longest prefix suffix match is?

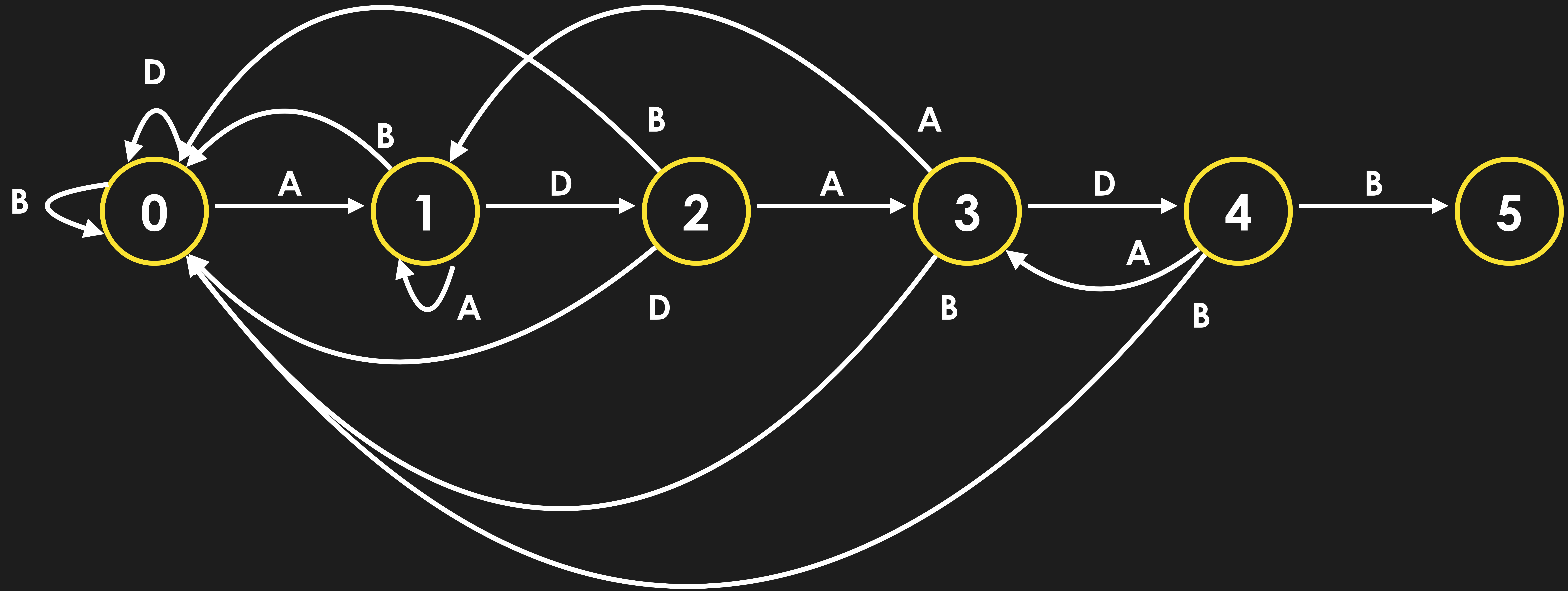
The Problem: How do we know what the longest prefix suffix match is?

By using a deterministic finite automaton (DFA)!

What is and how does a DFA work?

What is and how does a DFA work?

A DFA is a series of states, and a set of instructions on how to move between states



CodeIT

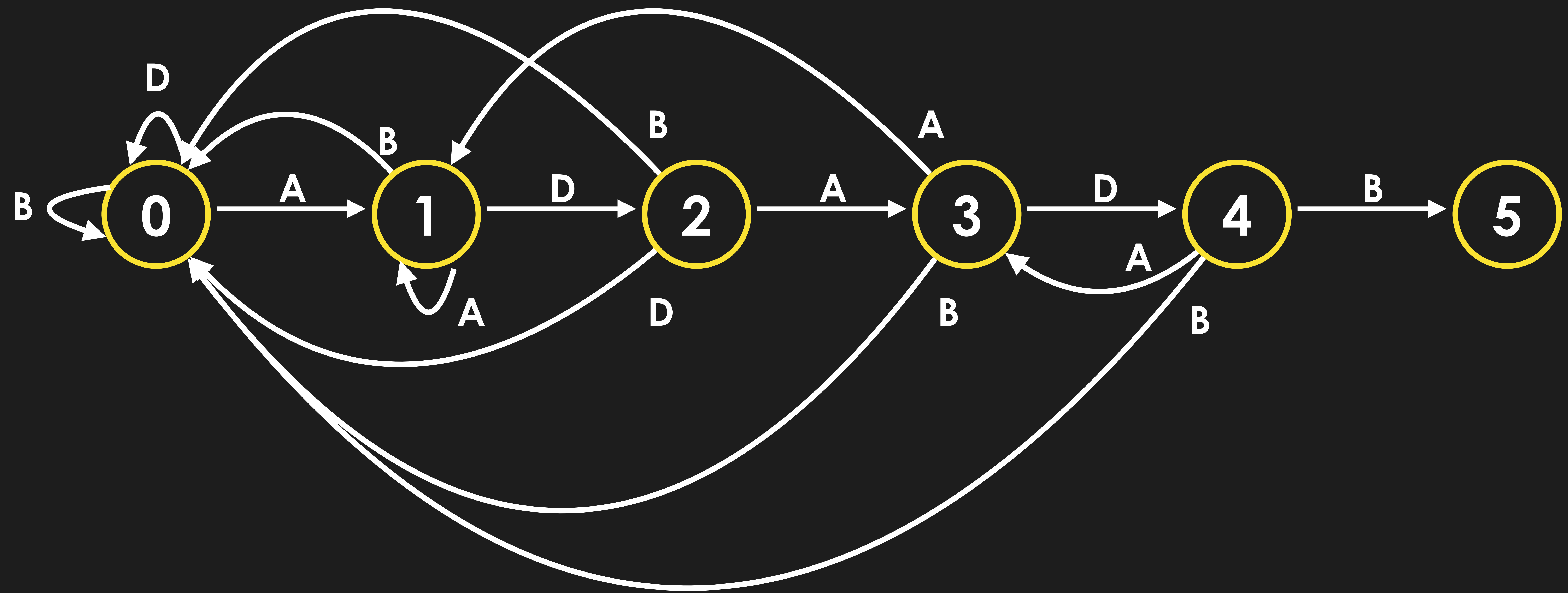
Example 1

text

A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4



text

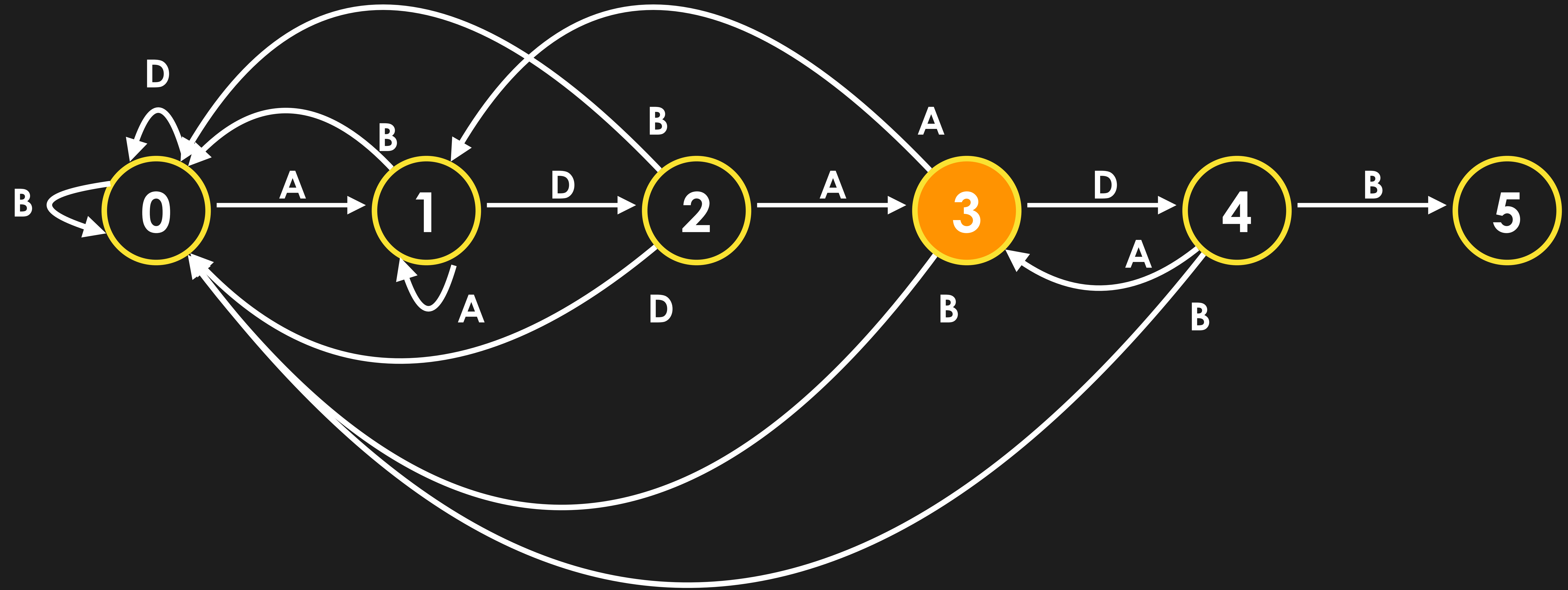
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Let's say we are at **state 3**
(matched 3 chars)



text

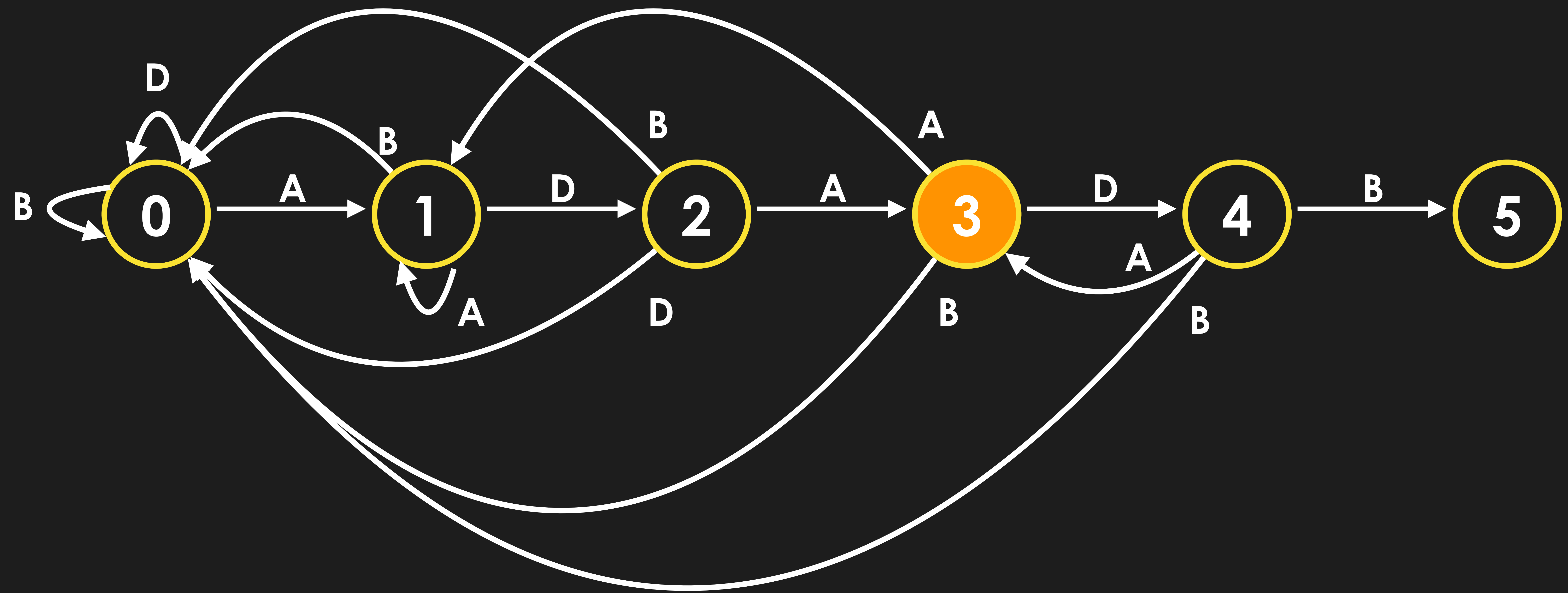
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Our next char is **'A'**. Regardless of whether it is a match or mismatch, we simply follow the DFA



text

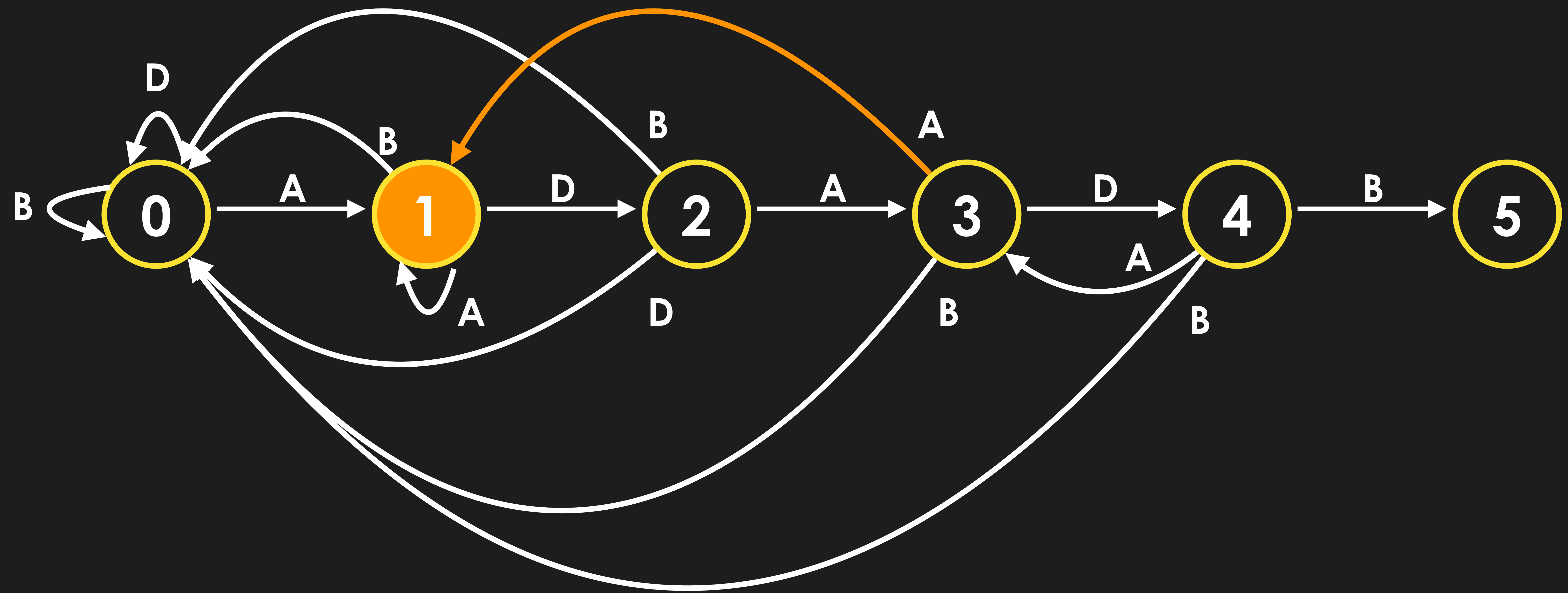
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Our next char is **'A'**. Regardless of whether it is a match or mismatch, we simply follow the DFA



text

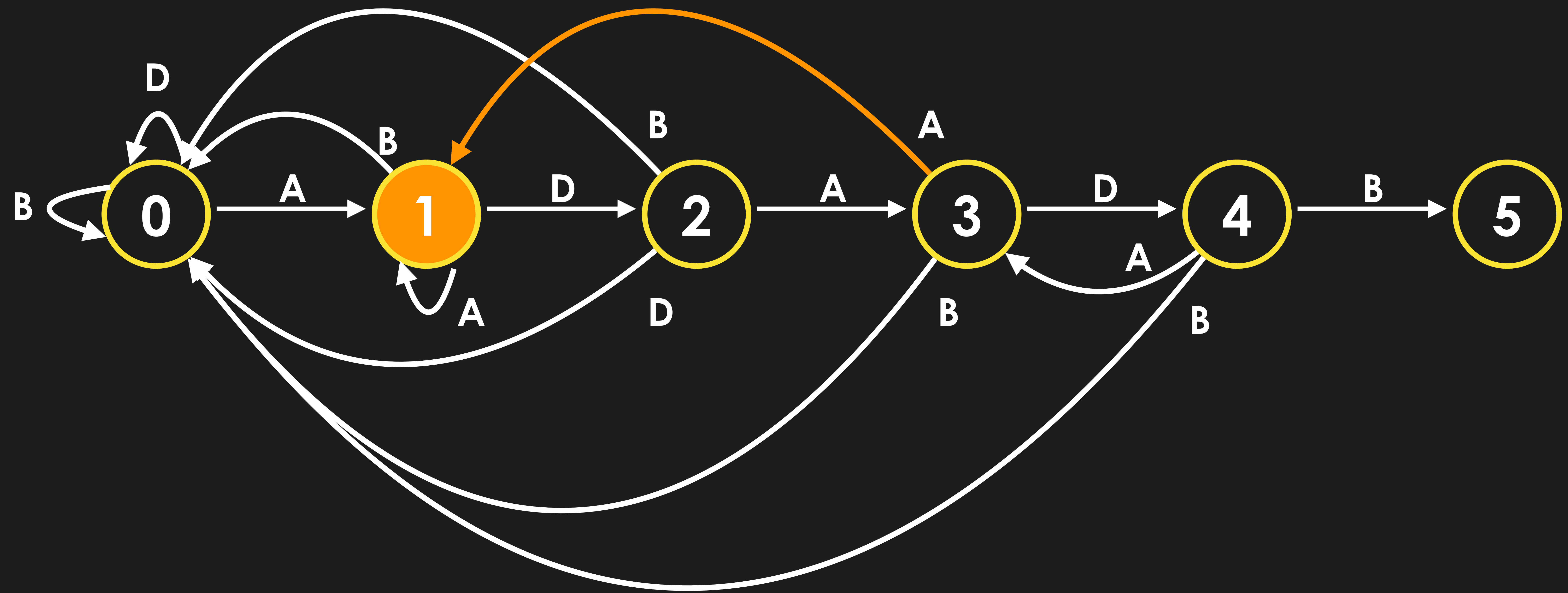
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Notice how the path that the DFA led us to is the longest prefix that is a suffix of the chars we have encountered



text

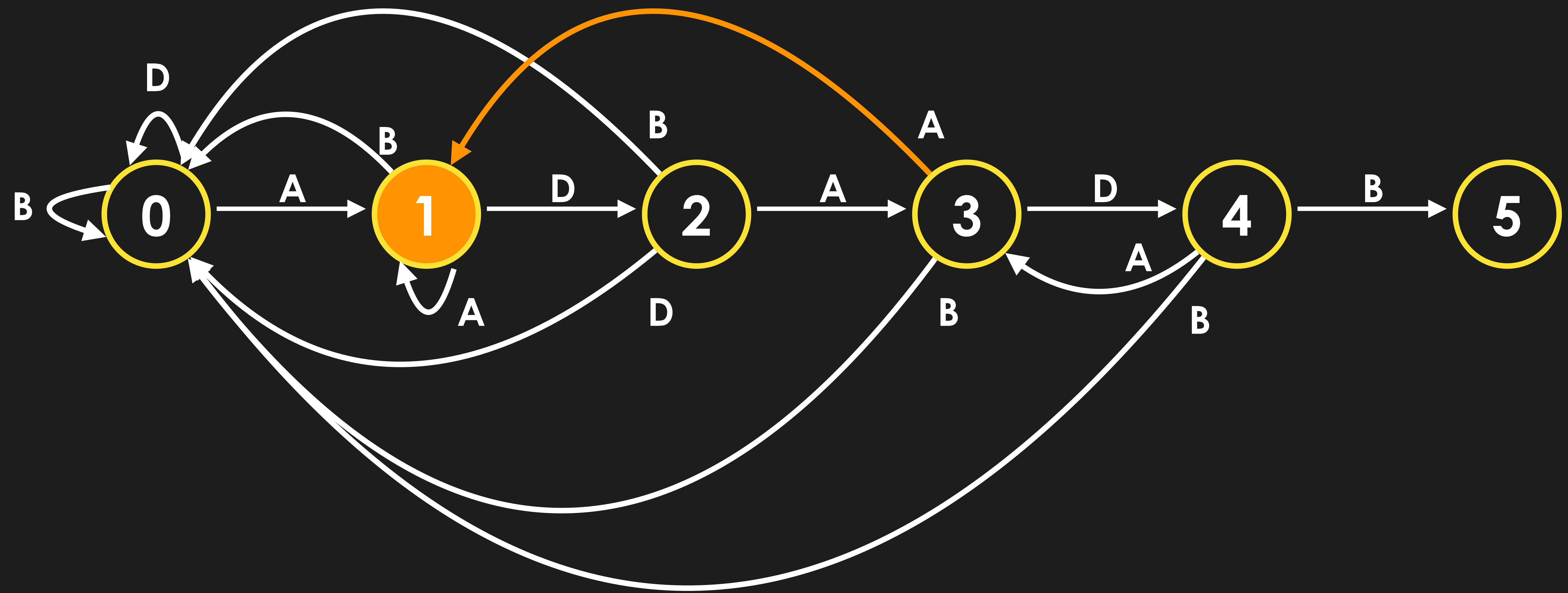
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Now we continue checking subsequent chars in text, moving state accordingly



text

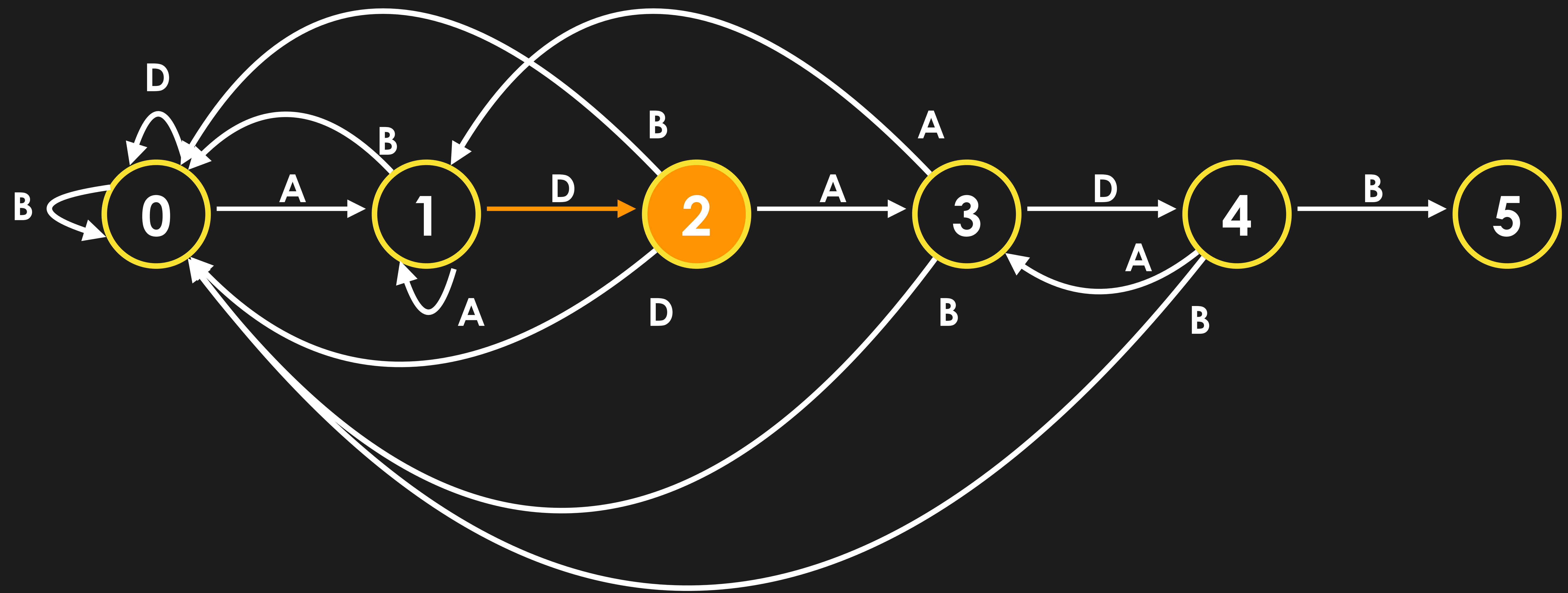
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Now we continue checking subsequent chars in text, moving state accordingly



text

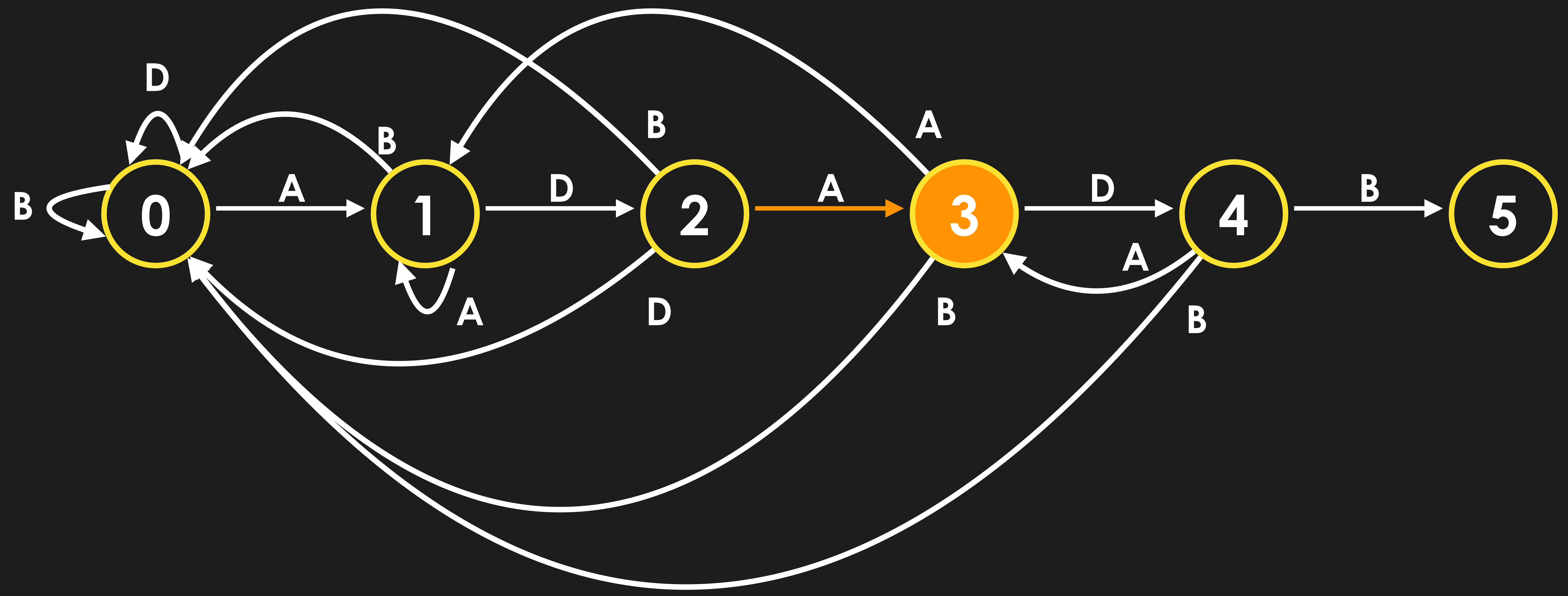
A	D	A	A	D	A	D

substring

A	D	A	D	B
0	1	2	3	4

Example

Now we continue checking subsequent chars in text, moving state accordingly



How do we represent a DFA? 2D List!

The **col number** tells us which **state** we are currently in
The **row number** indicates which character our current iteration of the text is
The **element** at **dfa[col][row]** tells us the next state

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

Demo of KMP using DFA

DFA Demo

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

DFA Demo

j indicates which state we are currently in

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	**1**	**2**	**3**	**4**

dfa

	0	**1**	**2**	**3**	**4**	**5**
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

CodeIT

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

Here, we have come across our first **mismatch.** Notice how we stay at state 1. This is because our DFA knows that **"A"** is the longest **suffix** match of a substring prefix!

dfa[row][col] indicates our **next state**

CodeIT

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

CodeIT

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

CodeIT

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

0	1	2	3	4
A	D	A	D	B

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

DFA Demo

j indicates which state we are currently in

i indicates which char of the text we are at

text

i

A	A	A	D	A	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

When j has reached the final state (length of substring), then we know that it has been matched!

dfa

	0	1	2	3	4	5
A	1	1	3	1	3	0
D	0	2	0	4	0	0
B	0	0	0	0	5	0

dfa[row][col] indicates our **next state**

CodeIT

Pointers about KMP

We only ever make **one pass over the original text**

A DFA is only dependent on the **substring** and is independent of the **text**

Why does a DFA work?

Why does a DFA work?

Imagine we have the following mismatch:

A	D	A	A	D	A	B	B	C

A	D	A	B	B

	0	1	2	3	4	5
A	1	1	3	1	1	0
D	0	2	0	2	0	0
B	0	0	0	4	5	0

Why does a DFA work?

Imagine we have the following mismatch:

A	D	A	A	D	A	B	B	C

A	D	A	B	B

	0	1	2	3	4	5
A	1	1	3	1	1	0
D	0	2	0	2	0	0
B	0	0	0	4	5	0

At this point, we don't have to **reset the entire check from the next index**

Why does a DFA work?

Imagine we have the following mismatch:

A	D	A	A	D	A	B	B	C

A	D	A	B	B

	0	1	2	3	4	5
A	1	1	3	1	1	0
D	0	2	0	2	0	0
B	0	0	0	4	5	0

Instead, based on our **DFA**, it knows that if we are at **state 3** and the mismatch is at **char D**, then we know that "**A**" (index 2 & 3) is the **longest matching suffix**, and so all we need to do is continue pattern matching from there

Why does a DFA work?

Imagine we have the following mismatch:

A	D	A	A	D	A	B	B	C

A	D	A	B	B

	0	1	2	3	4	5
A	1	1	3	1	1	0
D	0	2	0	2	0	0
B	0	0	0	4	5	0

Instead, based on our **DFA**, it knows that if we are at **state 3** and the mismatch is at **char D**, then we know that "**A**" (index 2 & 3) is the **longest matching suffix**, and so all we need to do is continue pattern matching from there

Knuth Morris Pratt Algorithm

Step 1: Build a **Deterministic Finite Automaton (DFA)** which guides the movement of our **iteration**

Step 2: Make one iteration over the text, keeping track of current **state** during each iteration

How to build a DFA?

How to build a DFA?

While it is possible to build a 2D list DFA for **KMP**, the process is tedious and complex

Instead, we are going to build a **compact version**, called a **"longest proper prefix which is also suffix"** table (LPS Table)

It is important to note that the mechanisms of this table are there to support what a DFA does

What is an LPS?

substring

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

An LPS tells us the longest prefix which is also a suffix

What is an LPS?

substring

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

An LPS tells us the longest prefix which is also a suffix

Take for example **lps[2]:**

- **substring[0:1]** is the longest prefix that is also a suffix **substring[2:3]**
- **"A" == "A"**

What is an LPS?

substring

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

An LPS tells us the longest prefix which is also a suffix

Take for example **lps[2]:**

- **substring[0:1]** is the longest prefix that is also a suffix **substring[2:3]**
- **"A" == "A"**

What is an LPS?

substring

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

An LPS tells us the longest prefix which is also a suffix

Looking at another example **lps[3]:**

- **substring[0:2]** is the longest prefix that is also a suffix **substring[2:4]**
- **"AD" == "AD"**

What is an LPS?

substring

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

An LPS tells us the longest prefix which is also a suffix

Looking at another example **lps[3]:**

- **substring[0:2]** is the longest prefix that is also a suffix **substring[2:4]**
- **"AD" == "AD"**

By knowing what is the longest prefix that is also a suffix, we can easily tell where to move j as we iterate through the text

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

Mismatch encounter, so we move j to **lps[j - 1]** (the longest prefix **possible**)

text

			i									
A	D	A	C	A	D	A	D	A	D	A	D	B

substring

			j	
A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

Mismatch encounter, so we move j to **lps[j - 1]** (the longest prefix **possible**)

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

Once again, we have a mismatch so we move j to **lps[j - 1]**. This is an example of how the lps simply **only** tracks the longest previous prefix **possible**

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

LPS

0	0	1	2	0
0	1	2	3	4

substring

j

A	D	A	D	B
0	1	2	3	4

LPS Demo

Once again, we have a mismatch so we move j to **lps[j - 1]**. This is an example of how the lps simply **only** tracks the longest previous prefix **possible**

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

Our prefix matches, so we can move on!

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

CodeIT

LPS Demo

text

										i		
A	D	A	C	A	D	A	D	A	D	A	D	B

substring

				j
A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

CodeIT

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

										i		
A	D	A	C	A	D	A	D	A	D	A	D	B

substring

		j		
A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

											i		
A	D	A	C	A	D	A	D	A	D	A	D	B

substring

			j	
A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

text

												i
A	D	A	C	A	D	A	D	A	D	A	D	B

substring

				j
A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

LPS Demo

Algorithm terminates when j increments past the length of the substring!

text

i

A	D	A	C	A	D	A	D	A	D	A	D	B

substring

j

A	D	A	D	B
0	1	2	3	4

LPS

0	0	1	2	0
0	1	2	3	4

CodeIT

How to build the LPS

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

A. Initialise length M array of 0s
B. Set j to be 0
C. Increment through substring with i:

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

CodeIT

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

A. Initialise length M array of 0s
B. Set j to be 0
C. Increment through substring with i:

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

A. Initialise length M array of 0s
B. Set j to be 0
C. Increment through substring with i:

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j (at index 1)

i (at index 2)

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	0	0	0	0
0	1	2	3	4	5	6	7

A. Initialise length M array of 0s
B. Set j to be 0
C. Increment through substring with i:

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

CodeIT

How to build the LPS

j

i

A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	0	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

		j					
				i			
A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	1	2	0	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j (at index 3)

i (at index 4)

substring

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

0	0	1	2	3	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	3	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	1	2	3	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	3	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

CodeIT

How to build the LPS

j

i

substring	A	D	A	D	A	A	D	A
	0	1	2	3	4	5	6	7

0	0	1	2	3	0	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j (at index 1)

i (at index 5)

A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	3	1	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

		j						
							i	
	A	D	A	D	A	A	D	A
	0	1	2	3	4	5	6	7

substring

0	0	1	2	3	1	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

	j							
						i		
A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	3	1	0	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	1	2	3	1	2	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

		j						
							i	
A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	3	1	2	0
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	1	2	3	1	2	0
0	1	2	3	4	5	6	7

A. Initialise length M array of 0s
B. Set j to be 0
C. Increment through substring with i:

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j

i

A	D	A	D	A	A	D	A	substring
0	1	2	3	4	5	6	7	

0	0	1	2	3	1	2	3
0	1	2	3	4	5	6	7

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

How to build the LPS

j (at index 3), i (at index 7)

A	D	A	D	A	A	D	A
0	1	2	3	4	5	6	7

substring

0	0	0	1	2	3	1	2	3
0	1	2	3	4	5	6	7	

A. **Initialise length M array of 0s**
B. **Set j to be 0**
C. **Increment through substring with i:**

1. While **j > 0** and **substring[j] != substring[i]:**
   - j = lps[j - 1]

2. If **substring[j] == substring[i]:**
   - lps[i] = j + 1

Note: In some implementations, memo might be stored like so, where everything is shifted forwards by one

Implementation of Knuth Morris Pratt

BuildLPS

```python
def buildLPS(substring):
    lps = [0] * len(substring)
    j = 0

    for i in range(1, len(substring)):
        while j > 0 and substring[i] != substring[j]:
            j = lps[j - 1]

        if substring[j] == substring[i]:
            j += 1
            lps[i] = j

    return lps
```

KMP

```python
def KnuthMorrisPratt(text, substring):
    lps = buildLPS(substring)

    i = 0
    j = 0

    for i in range(len(text)):
        while (j > 0 and text[i] != substring[j]):
            j = lps[j - 1]
        if text[i] == substring[j]:
            j += 1
        if j == len(substring):
            print("Found match at index {}".format(i - len(substring) + 1))
            j = lps[j - 1]
```

Analysis of KMP

Time complexity of DFA

1. The time complexity of **constructing** a substring DFA:

   R (number of unique chars in substring) * K (length of substring)

2. The time complexity of searching for a substring within a text using the DFA takes:

   N (length of text)

Time complexity of LPS

1. The time complexity of **constructing** a substring LPS:

   K (length of substring)

2. The time complexity of searching for a substring within a text using the DFA takes:

   N (length of text)

Although the LPS table is easier to construct than the DFA, it does create some overhead when iterating over the string because the lps table doesn't provide a **guaranteed match when shifting j**

Time complexity of KMP

1. Overall, the time complexity of KMP is **O(K + N)**

2. It is important to note that computing the LPS / DFA only needs to be **done once for a substring**, and can be used repeatedly for any text

CodeIT

Difference between DFA and LPS

1. Using the DFA, we are given a guaranteed match after state transition, whereas, for LPS, we are directed to the longest possible prefix given that mismatch, and may make up to K transitions

2. In the DFA, we consider **current state, and current char**. For LPS, we only consider **match or mismatch**.

4. LPS is a more popular implementation of the KMP algorithm

Lab Session 1

Lab Session 1

- In this lab session, you will be implementing kmp**.py**
- Your task is to implement **Knuth Morris Pratt** algorithm, using a **longest prefix suffix (LPS) table**
- You should implement the **buildLPS & KnuthMorrisPratt** functions
- **buildLPS** takes in a substring and returns a length M, where M is len(substring), array representing the lps table for that substring
- **KnuthMorrisPratt** takes in a text & substring, and returns a list of all indices marking the start of a substring match in the text
- To test, run `**python utils/kmp_test.py**`

Solution: buildLPS

```python
def buildLPS(substring):
    lps = [0] * len(substring)
    j = 0

    for i in range(1, len(substring)):
        while j > 0 and substring[i] != substring[j]:
            j = lps[j - 1]

        if substring[j] == substring[i]:
            j += 1
            lps[i] = j

    return lps
```

Solution: KnuthMorrisPratt

```python
def KnuthMorrisPratt(text: str, substring: str):

    lps = buildLPS(substring)

    j = 0
    res = []
    for i in range(len(text)):
        while (j > 0 and text[i] != substring[j]):
            j = lps[j - 1]
        if text[i] == substring[j]:
            j += 1
        if j == len(substring):
            res.append(i - len(substring) + 1)
            j = lps[j - 1]

    return res
```

Can we do better?

Can we do better?

- Brute-Force: **O(N * M)**
- KMP: **O(N)**
- Boyer-Moore: ?

What is Boyer-Moore?

KMP vs Boyer-Moore

A	D	A	C	A	D	A	D	A	D	A	D	B	**text**

A	D	A	D	B	**substring**
0	1	2	3	4	

While KMP matches chars forwards, and remembers the **longest prefix,** such that you only ever **iterate once through the text**

KMP vs Boyer-Moore

A	D	A	C	A	D	A	D	A	D	A	D	B	**text**

A	D	A	D	B	**substring**
0	1	2	3	4	

Boyer Moore starts with comparisons backwards!

KMP vs Boyer-Moore

A	D	A	C	A	D	A	D	A	D	A	D	B	text

A	D	A	D	B	substring
0	1	2	3	4	

Boyer Moore starts with comparisons backwards!

KMP vs Boyer-Moore

A	D	A	C	A	D	A	D	A	D	A	D	B	**text**

A	D	A	D	B	**substring**
0	1	2	3	4	

When there's a mismatch, you simply move the substring forward based on two heuristics: **Bad Char Heuristic / Good Suffix Heuristic**

KMP vs Boyer-Moore

A	D	A	C	A	D	A	D	A	D	A	D	B	**text**

A	D	A	D	B	**substring**
0	**1**	**2**	**3**	**4**	

When there's a mismatch, you simply move the substring forward based on two heuristics: **Bad Char Heuristic / Good Suffix Heuristic**

In this lesson, we will only **cover the bad char heuristic**

Boyer-Moore Algorithm: Bad Char Heuristic

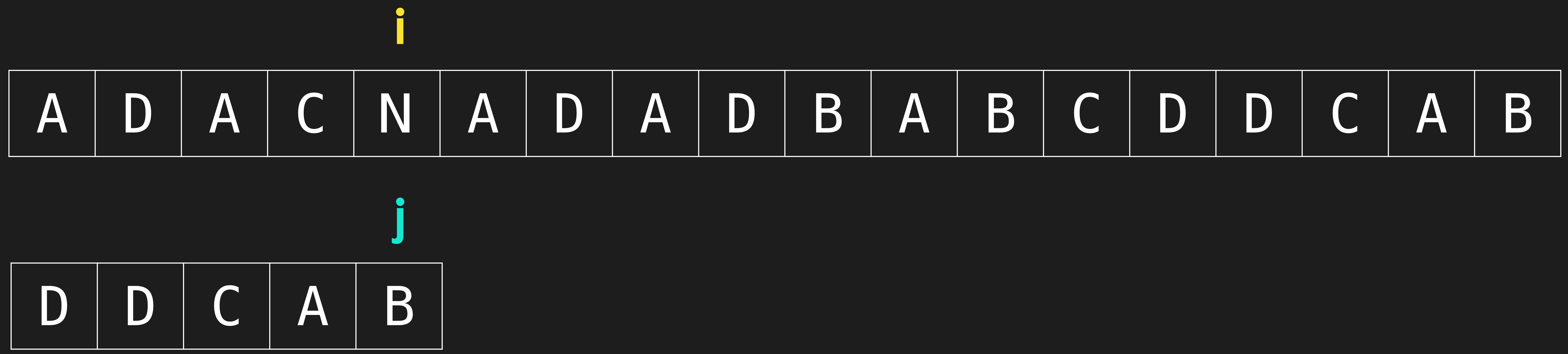
Boyer-Moore

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B	**text**

D	D	C	A	B	**substring**

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 1: Mismatch is not in pattern
- Shift pattern **M (length of substring) chars forward**

				i													
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

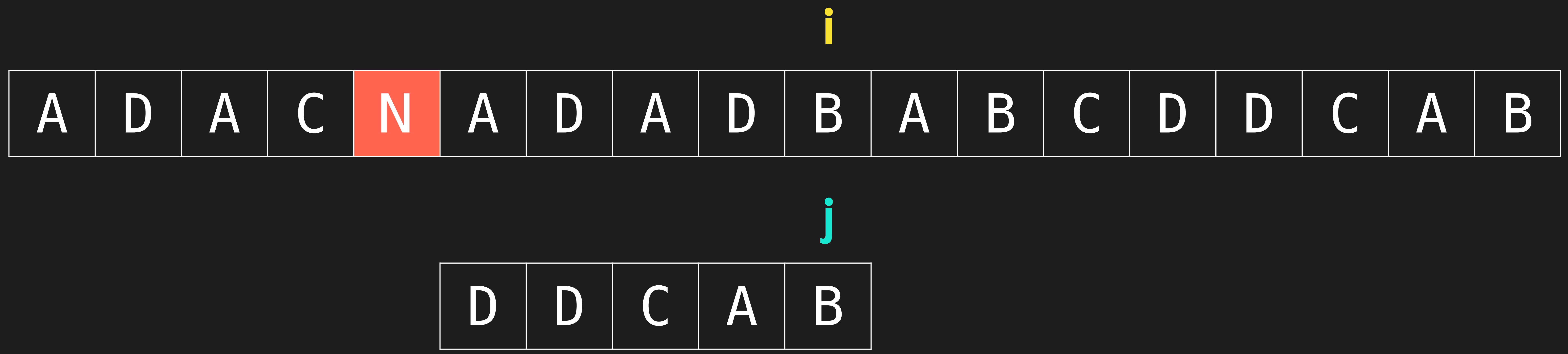
				j
D	D	C	A	B

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 1: Mismatch is not in pattern

- Shift pattern **M (length of substring) chars forward**



Boyer-Moore

- **i** indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- **j** indicates how far we've matched

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 2a: Mismatch is in pattern, move substring over to **last instance**

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

CodeIT

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 2a: Mismatch is in pattern, move substring over to **last instance**

											i						
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

Boyer-Moore

- **i** indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- **j** indicates how far we've matched

i

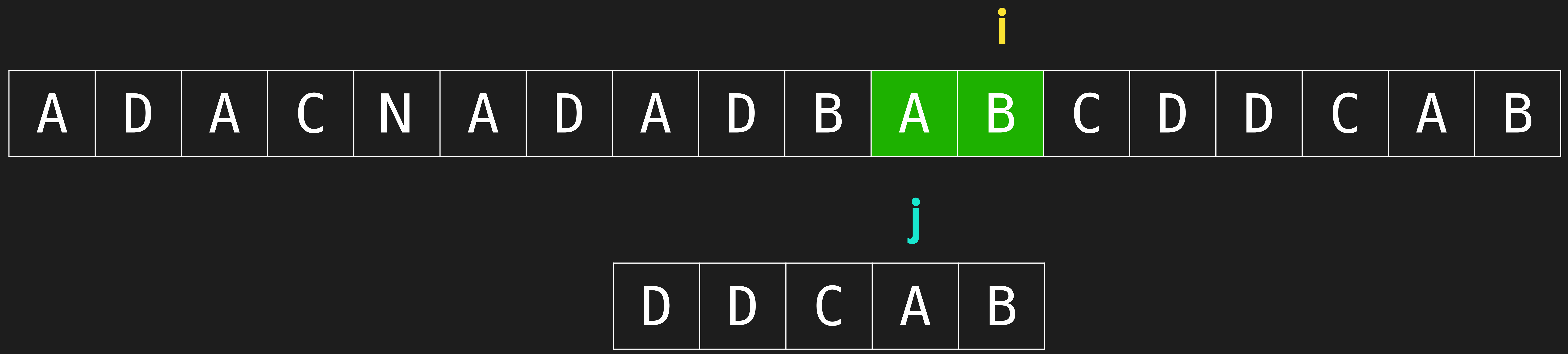
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

Boyer-Moore

- **i** indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- **j** indicates how far we've matched



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 2b: Mismatch is in pattern, but requires backtrack

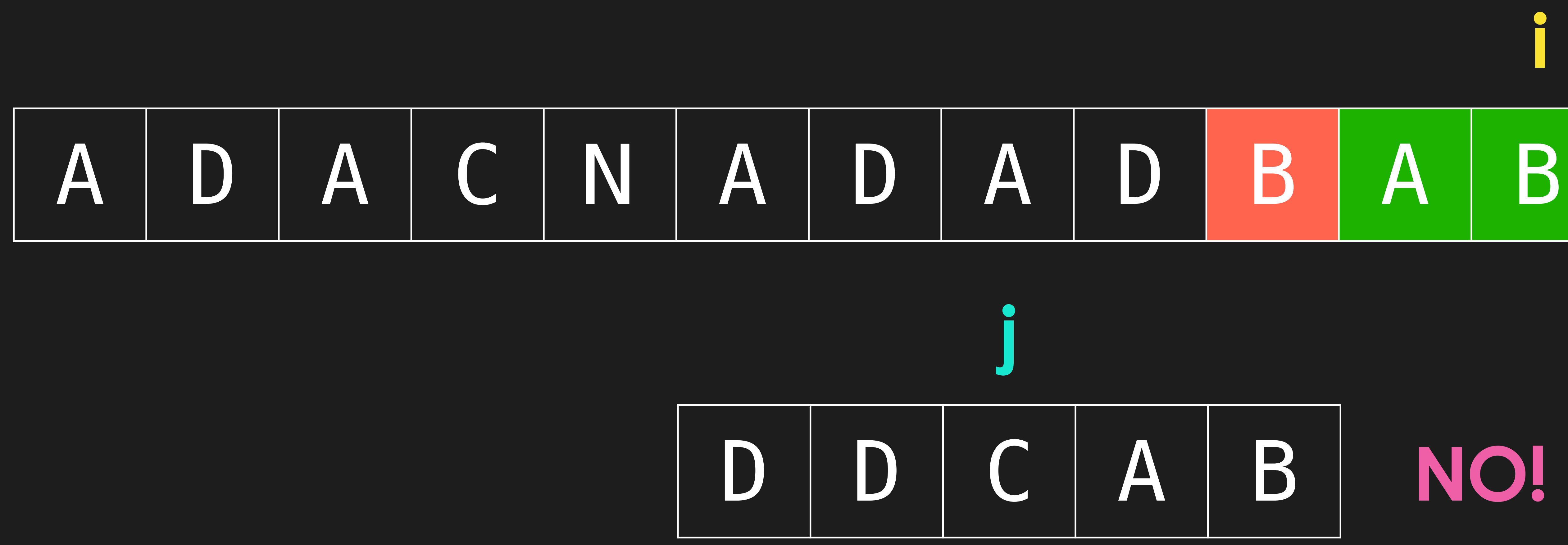
											i						
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

		j		
D	D	C	A	B

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 2b: Mismatch is in pattern, but requires backtrack



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 2b: Mismatch is in pattern, but requires backtrack (simply move one step forward)

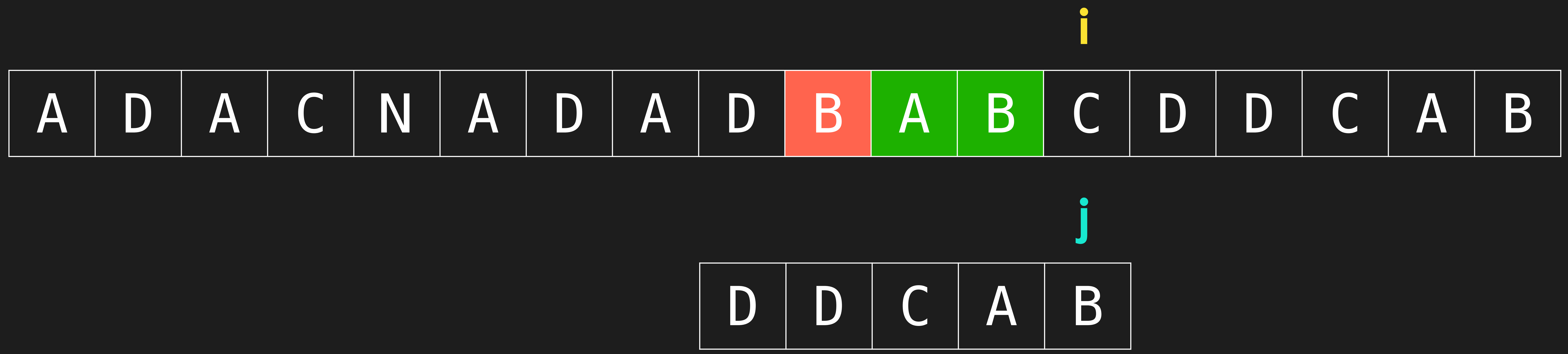
											i						
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

		j		
D	D	C	A	B

Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

Mismatch case 2b: Mismatch is in pattern, but requires backtrack (simply move one step forward)



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

i

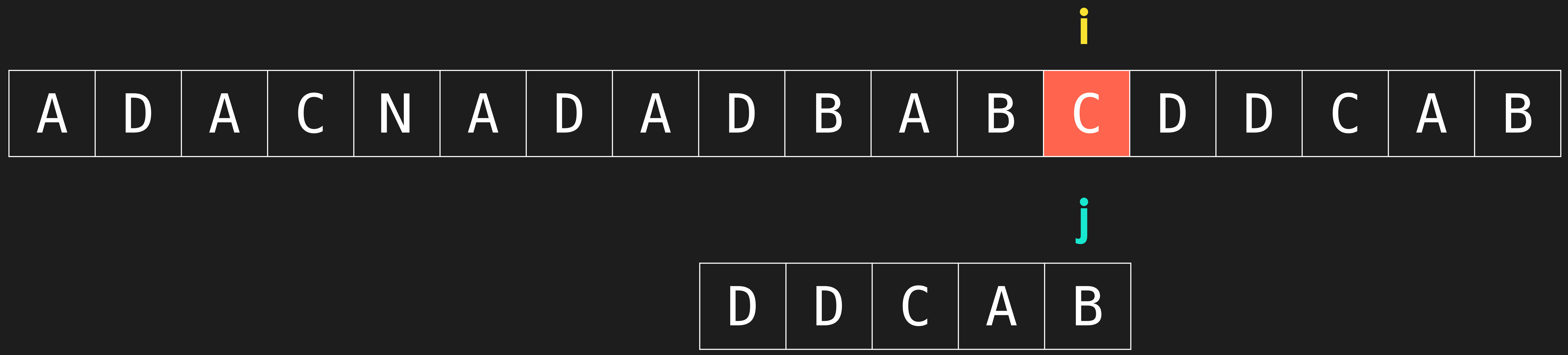
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

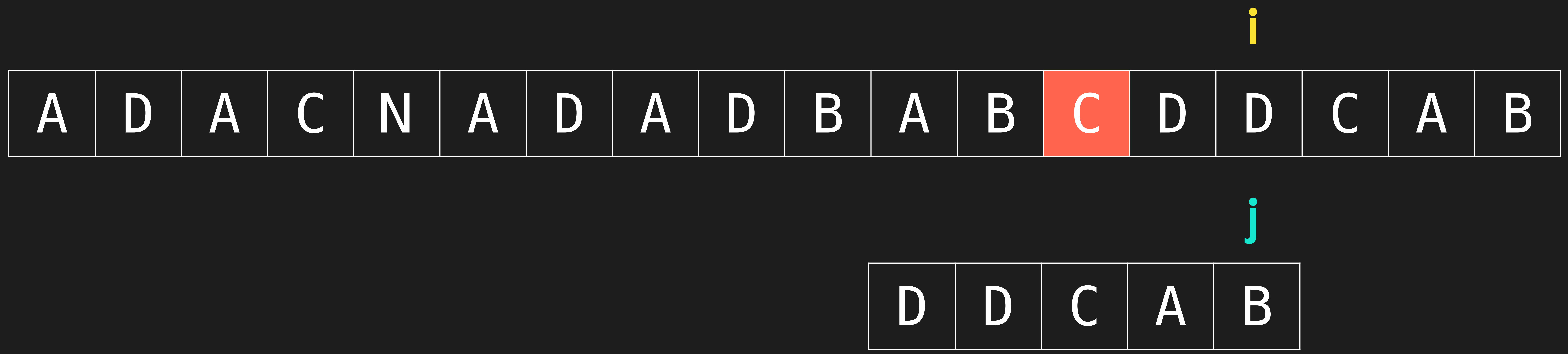
Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

i

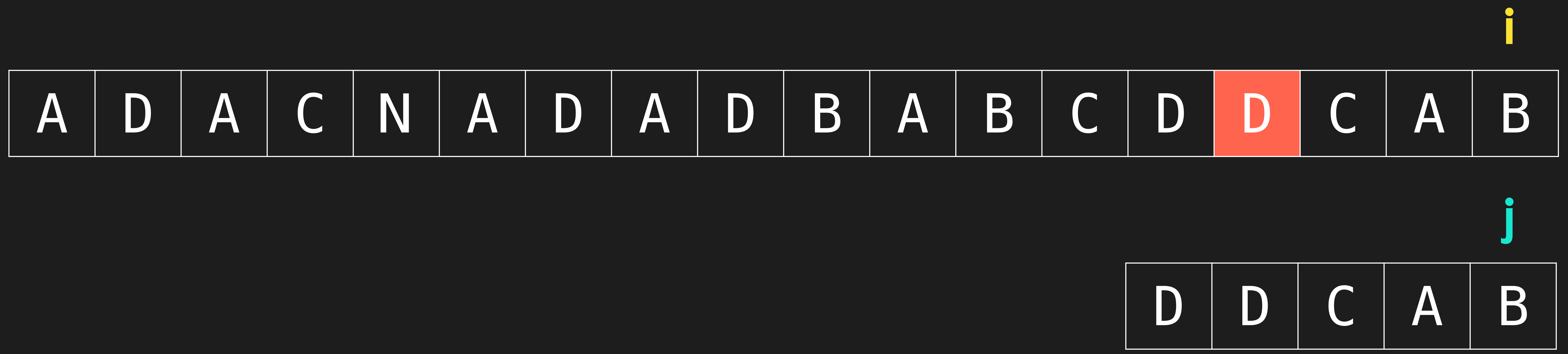
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

Boyer-Moore

- **i** indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched



Boyer-Moore

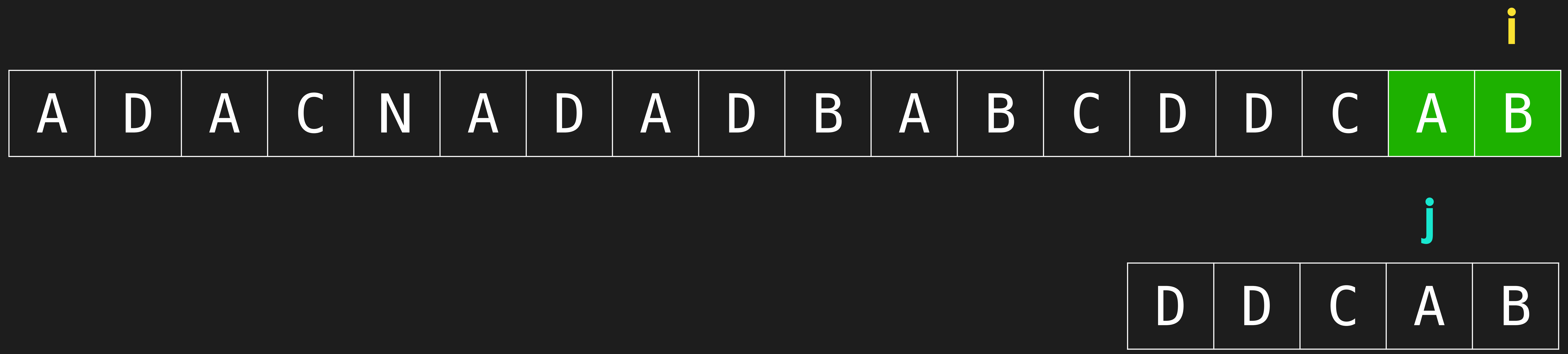
- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

																	i
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

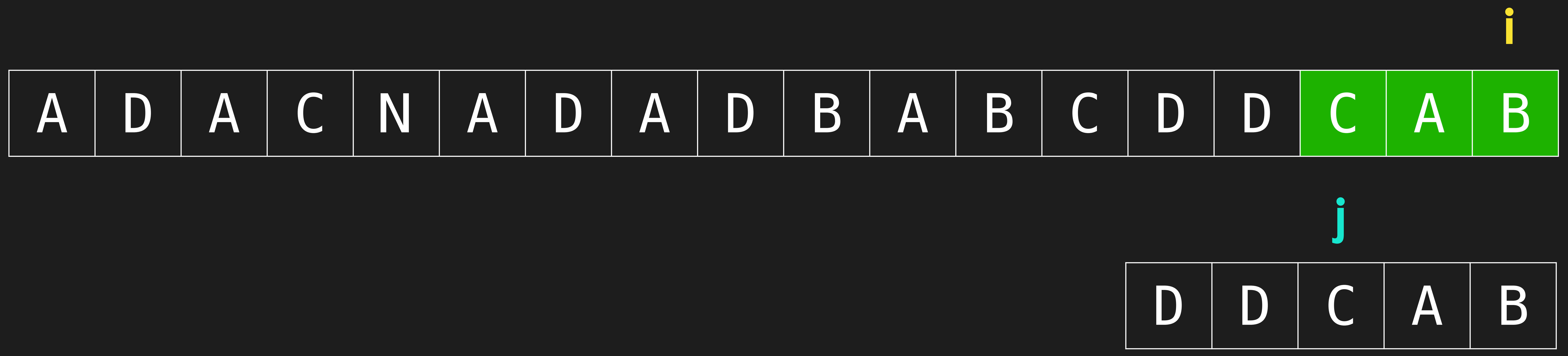
Boyer-Moore

- **i** indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- **j** indicates how far we've matched



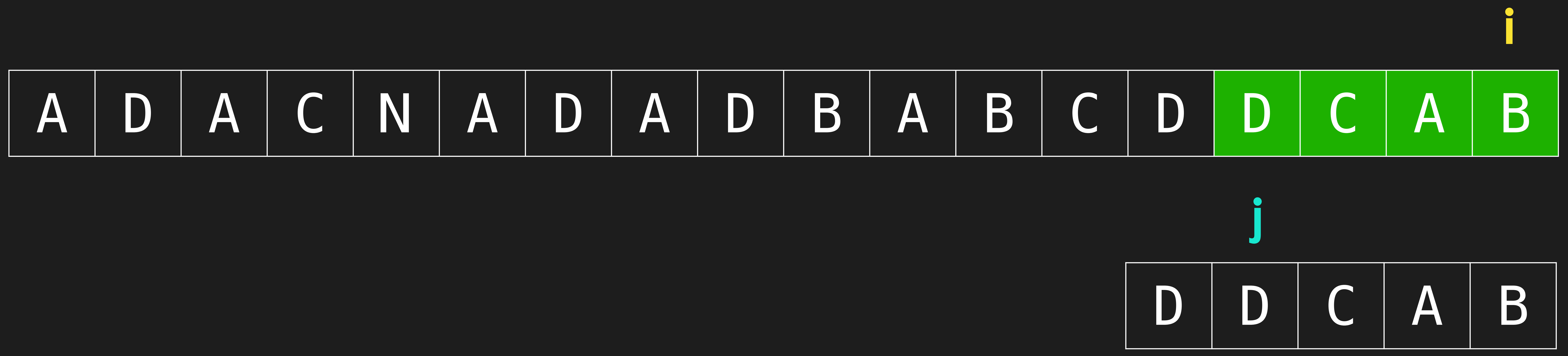
Boyer-Moore

- **i** indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- **j** indicates how far we've matched



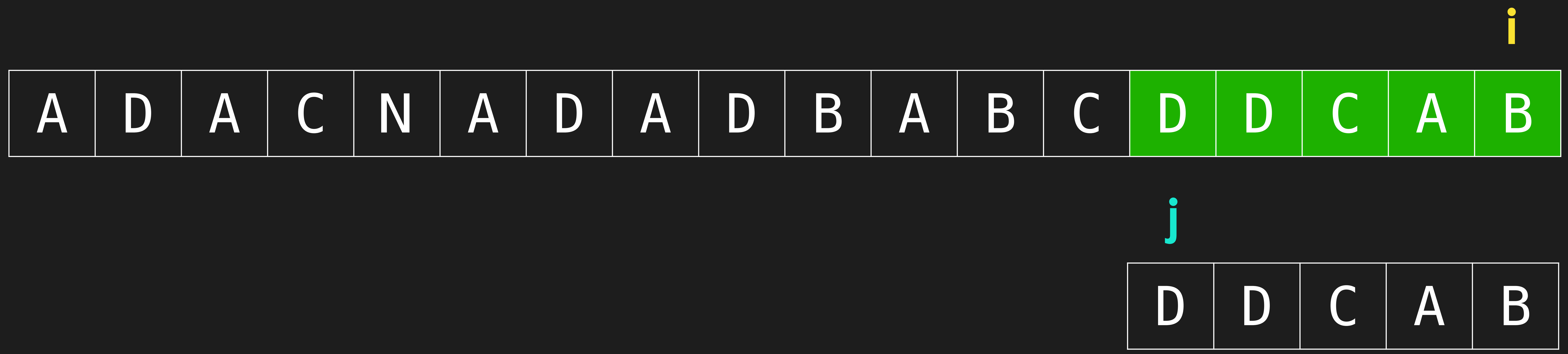
Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched



Boyer-Moore

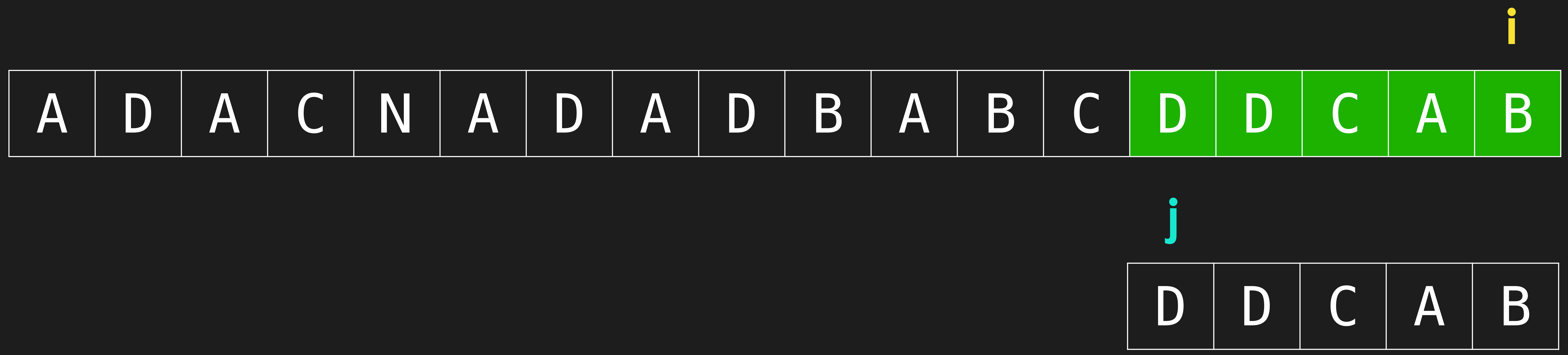
- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched



Boyer-Moore

- i indicates **where the matching begins** (we always match right to left in Boyer-Moore)
- j indicates how far we've matched

When j is at 0 (start of substring), it means we have matched all chars and substring is found!



How to determine last instance?

Creating a bad char heuristic table

D	D	C	A	B

lastAt

A	-1
B	-1
C	-1
D	-1

Creating a bad char heuristic table

Iterate through substring and record index of last instance of each char

D	D	C	A	B

lastAt

A	-1
B	-1
C	-1
D	-1

Creating a bad char heuristic table

Iterate through substring and record index of last instance of each char

D	D	C	A	B

j

lastAt

A	-1
B	-1
C	-1
D	0

Creating a bad char heuristic table

Iterate through substring and record index of last instance of each char

D	D	C	A	B
	j			

lastAt

A	-1
B	-1
C	-1
D	1

CodeIT

Creating a bad char heuristic table

Iterate through substring and record index of last instance of each char

D	D	C	A	B

j

lastAt

A	-1
B	-1
C	2
D	1

Creating a bad char heuristic table

Iterate through substring and record index of last instance of each char

D	D	C	A	B
			j	

lastAt

A	3
B	-1
C	2
D	1

CodeIT

Creating a bad char heuristic table

Iterate through substring and record index of last instance of each char

D	D	C	A	B
				j

lastAt

A	3
B	4
C	2
D	1

Using the lastAt table

CodeIT

Boyer-Moore

				i													
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

Boyer-Moore

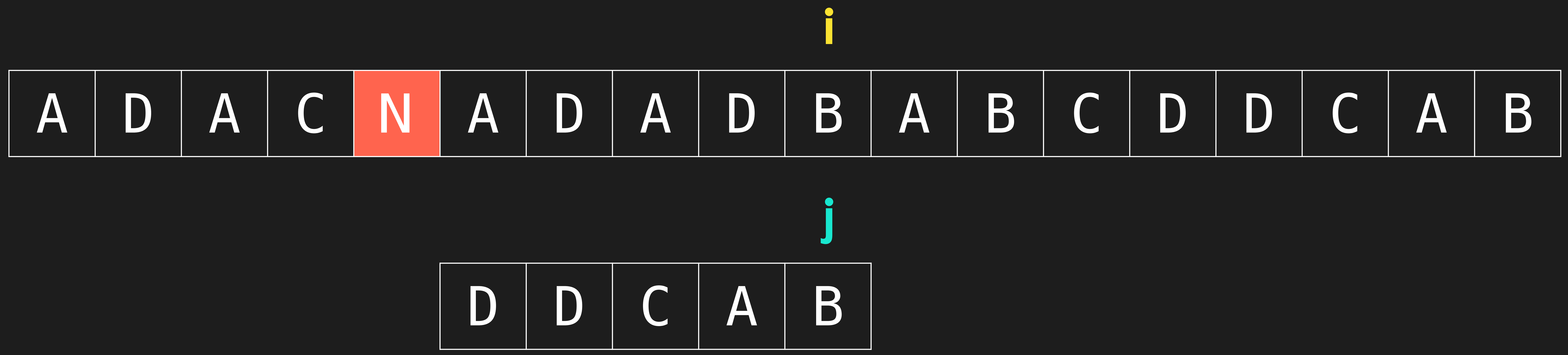
				i													
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

Boyer-Moore



lastAt

A	3
B	4
C	2
D	1

Boyer-Moore

									i								
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

On mismatch, move i over by:

```
j – lastAt[text[j]]
```

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

On mismatch, move i over by:

`j - lastAt[text[j]] = 3 - 1 = 2`

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

On mismatch, move i over by:

`j - lastAt[text[j]] = 3 - 1 = 2`

Boyer-Moore

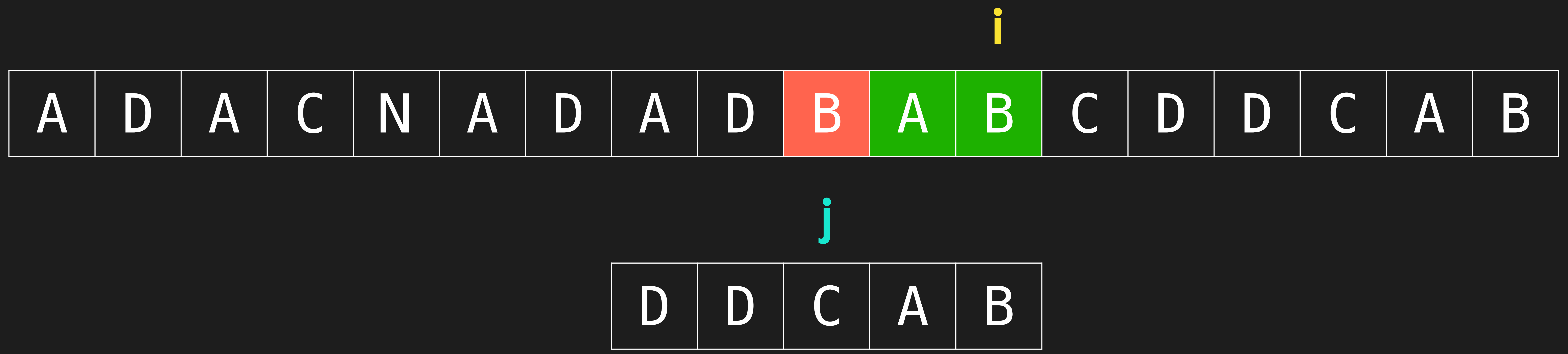
											i						
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

Boyer-Moore



lastAt

A	3
B	4
C	2
D	1

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

$$j - lastAt[text[j]] = 2 - 4 = -2$$

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

$$j - lastAt[text[j]] = 2 - 4 = -2$$

If negative, move by 1 only

CodeIT

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

`j - lastAt[text[j]] = 2 - 4 = -2`

If negative, move by 1 only

Boyer-Moore

												i					
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

Boyer-Moore

												i					
A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

				j
D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

```
j - lastAt[text[j]] = 4 - 2 = 2
```

Boyer-Moore

i

A	D	A	C	N	A	D	A	D	B	A	B	C	D	D	C	A	B

j

D	D	C	A	B

lastAt

A	3
B	4
C	2
D	1

```
j - lastAt[text[j]] = 4 - 2 = 2
```

Implementation of Boyer-Moore

badCharHeuristic

```python
def badCharHeuristic(substring):
    lastAt = {}


    for i in range(len(substring)):
        lastAt[substring[i]] = i

    return lastAt
```

BoyerMoore

```python
def BoyerMoore(string, substring):
    lastAt = badCharHeuristic(substring)

    M = len(substring)
    N = len(string)
    s = 0

    while (s <= N-M):
        j = M - 1

        while j >= 0 and string[s + j] == substring[j]:
            j -= 1

        if j < 0:
            print('pattern occured at index: {}'.format(s))
            s += 1
        else:
            s += max(1, j - lastAt[string[s + j]])
```

Analysis of Boyer-Moore Algorithm

Analysis of Boyer-Moore Algorithm

- On average, since we are matching from right to left, we are able to **skip M characters at each check**

- This means that **Boyer-Moore** has an average time complexity of **O (N/M)** !

- The longer the pattern, the faster Boyer-Moore becomes

Lab Session 2

Lab Session 2

- In this lab session, you will be implementing **boyer.py**
- Your task is to implement **Boyer Moore** algorithm, using the **bad char heuristic**
- You should implement the **badCharHeuristic** & **BoyerMoore** functions
- **buildLPS** takes in a substring and returns a dictionary, where dict[char] is equals to the last index at which the char is seen in the substring
- **BoyerMoore** takes in a text & substring, and returns a list of all indices marking the start of a substring match in the text
- To test, run `**python utils/boyer_test.py**`

Solution: badCharHeuristic

```python
def badCharHeuristic(substring: str):

    lastAt = {}

    for i in range(len(substring)):
        lastAt[substring[i]] = i

    return lastAt
```

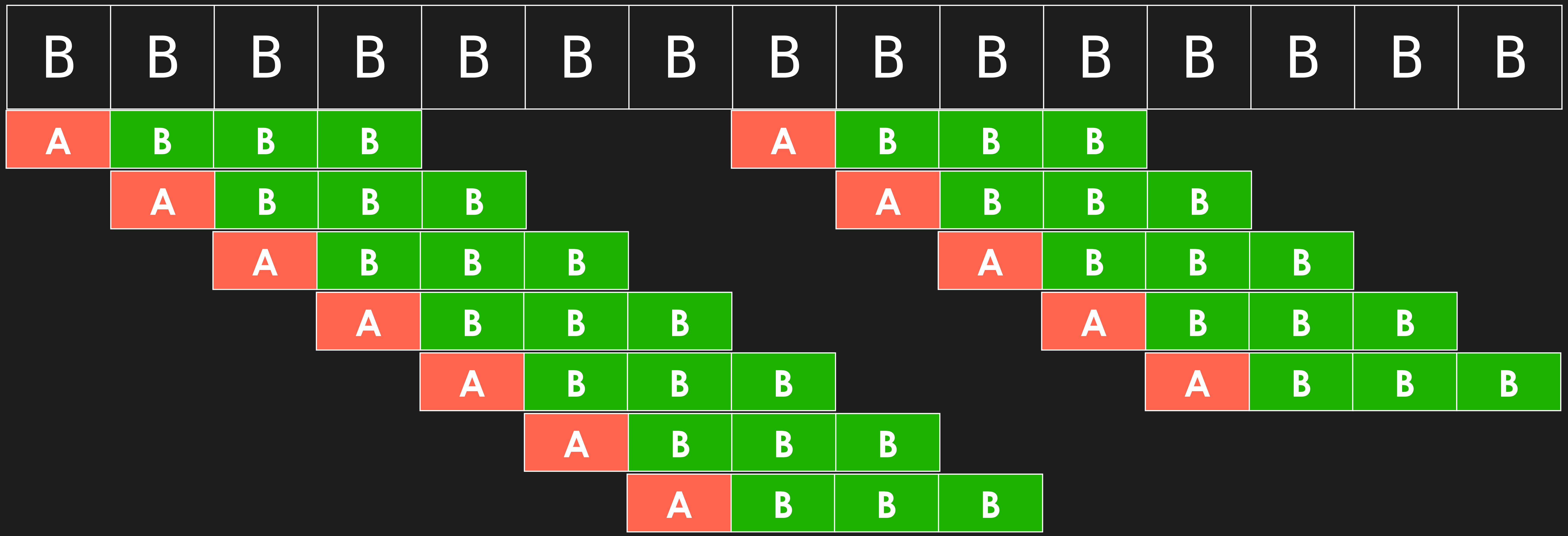
Solution: BoyerMoore

```python
def BoyerMoore(text: str, substring: str):

    lastAt = badCharHeuristic(substring)

    M = len(substring)
    N = len(text)
    s = 0
    res = []

    while (s <= N-M):
        j = M - 1

        while j >= 0 and text[s + j] == substring[j]:
            j -= 1

        if j < 0:
            res.append(s)
            s += 1

        else:
            s += max(1, j - lastAt[text[s + j]]) if text[s + j] in lastAt else 1

    return res
```

Caveat!

Boyer-Moore Worst Case: O(N * M)!

substring: "ABBB"



We check M - 1 times for each char in the text, hence we get N * M complexity!

CodeIT

Caveat

In the worst case, Boyer-Moore has **O (N * M)** complexity, the same as the brute force approach

This can be reduced however, if we use a strategy similar to KMP, where we keep track of the longest proper suffix!

Summary

CodeIT

	Brute-Force	KMP	Boyer-Moore (bad char heuristic only)
Average Case	N	N	**N / M**
Worst Case	M * N	**N**	M * N

CodeIT

	Brute-Force	KMP	Boyer-Moore (bad char heuristic only)
Average Case	N	N	**N / M**
Worst Case	M * N	**N**	M * N

Typically: Boyer-Moore will provide the fastest substring search efficiency

Guarantee: KMP provides the best guarantee of substring search efficiency