LECTURE 14

KNUTH–MORRIS–PRATT ALGORITHM

Big-O Coding

Website: www.bigocoding.com

Bài toán minh họa

Tìm kiếm chuỗi là bài toán cho một văn bản T (text) và một chuỗi mẫu P (pattern), hãy tìm kiếm vị trí xuất hiện của chuỗi P trong chuỗi T.

Ví dụ:

- Chuỗi T: “ABABAABACDABABCABAB”
- Chuỗi P: “ABABCABAB”

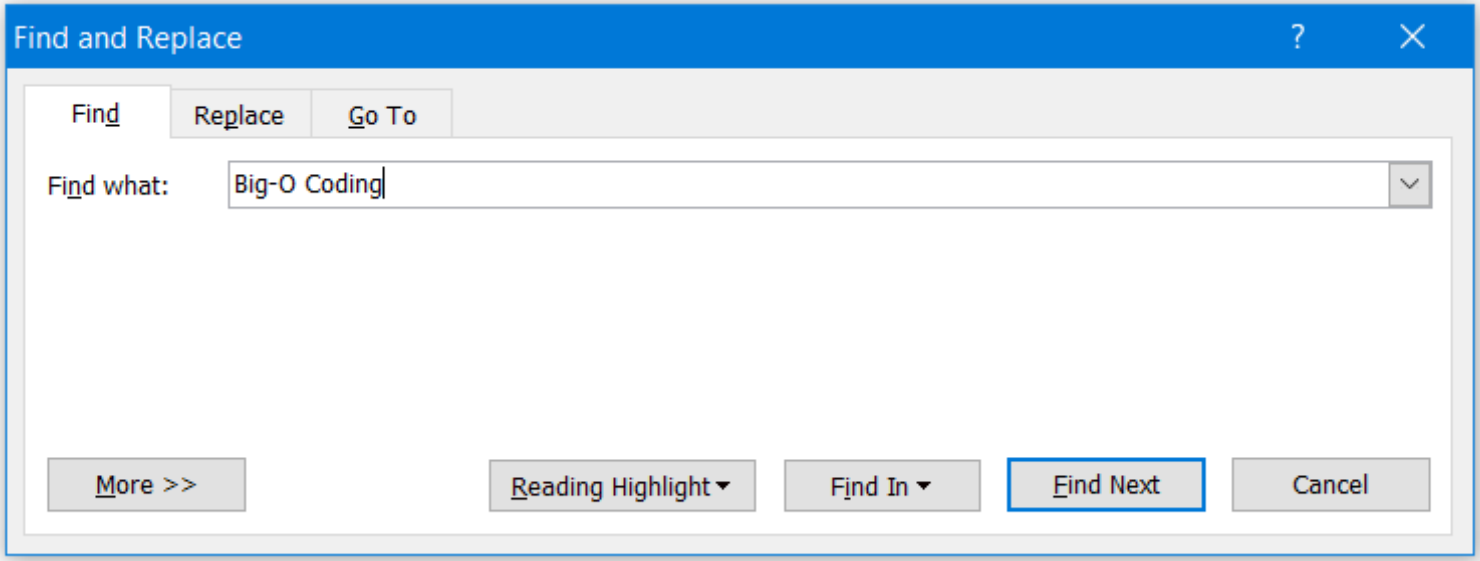
Tìm những vị trí chuỗi P xuất hiện trong chuỗi T.

➔ Chuỗi P xuất hiện trong T: “ABABAABACDABABCABAB“

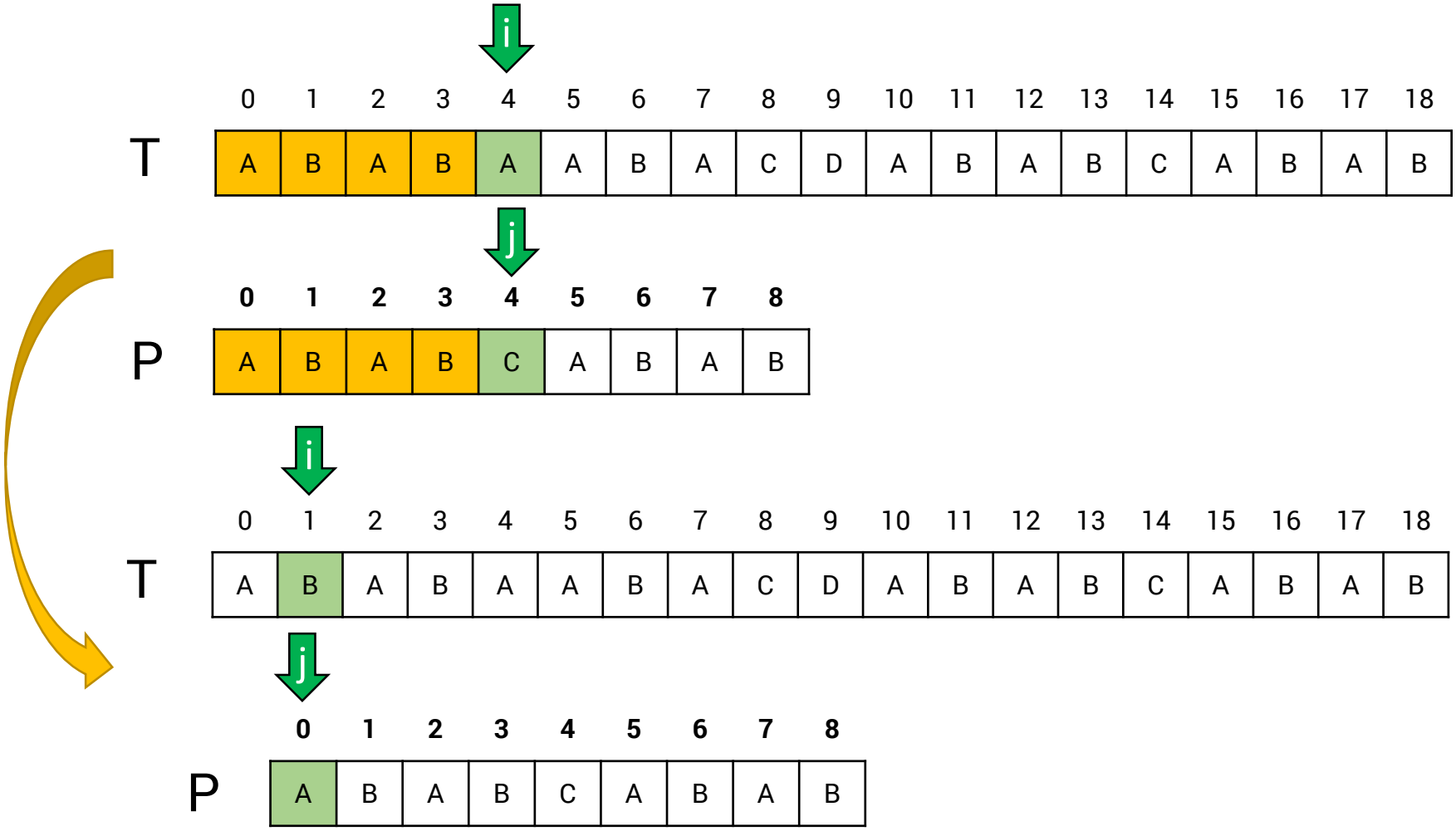
Vị trí số: 10

Bài toán minh họa

Có nhiều thuật toán để giải quyết bài toán này từ đơn giản đến phức tạp: Brute Force, Knuth–Morris–Pratt (KMP), Boyer-Moore, Z Function…



Giải bài toán bằng Brute Force

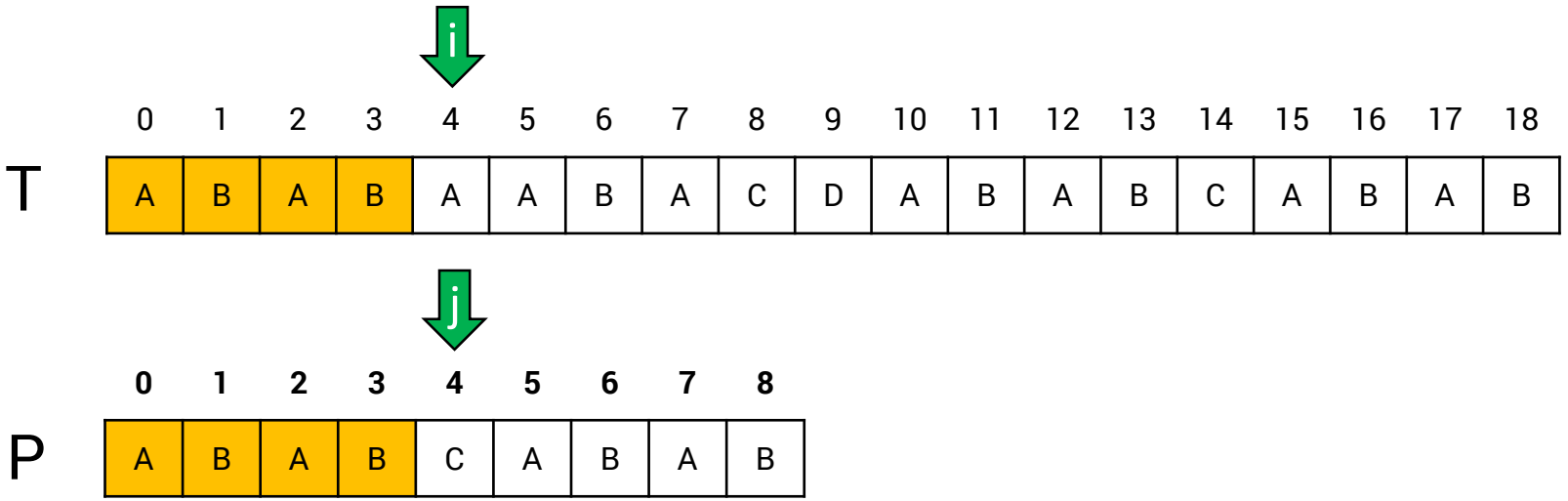


Time Complexity: O(N*M)

- N là độ dài của chuỗi T.
- M là độ dài của chuỗi P.

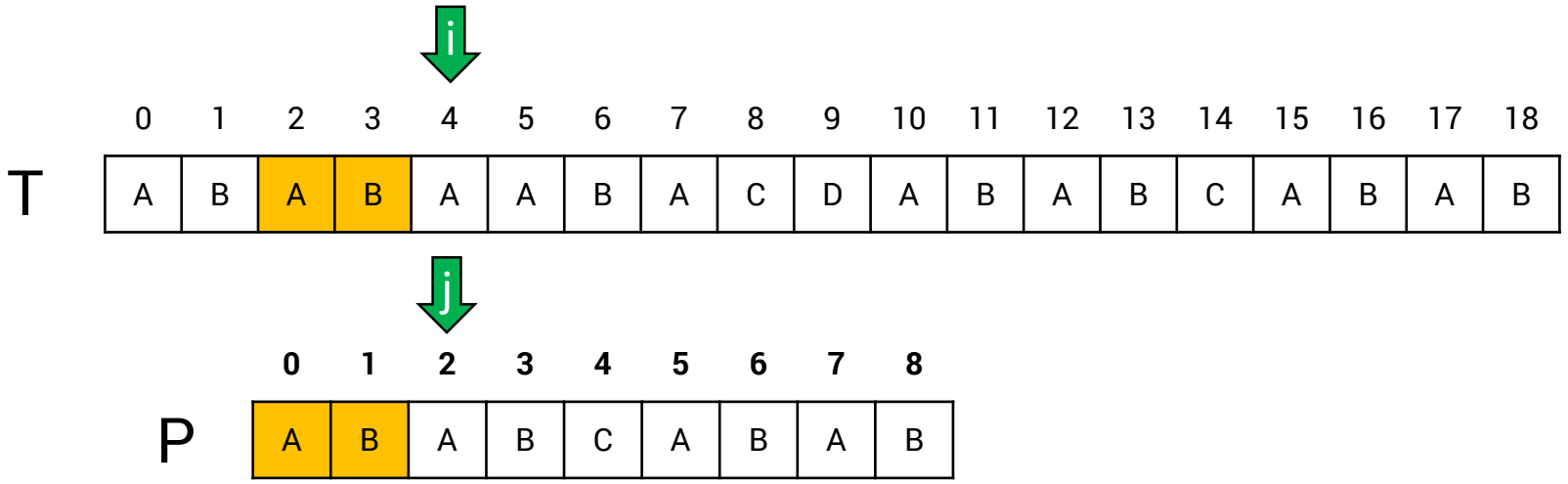
Thuật toán Knuth–Morris–Pratt

Knuth–Morris–Pratt (**KMP**): là một thuật toán tìm kiếm chuỗi với độ phức tạp tuyến tính **O(N + M)** được giới thiệu vào năm 1977.

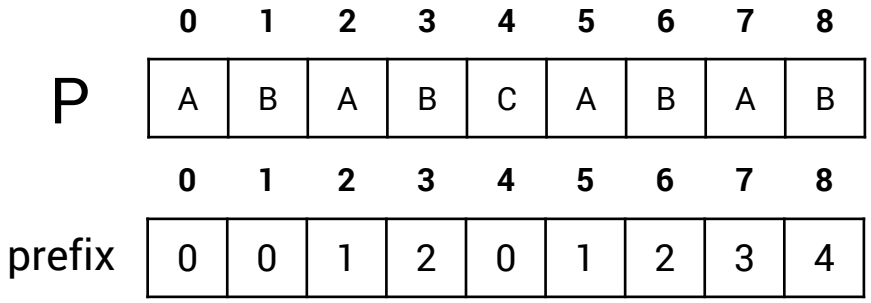


Ý tưởng: Khi gặp cặp ký tự không trùng khớp, thay vì dịch chuyển biến chạy của chuỗi T sang phải 1 đơn vị và so sánh lại từ đầu với chuỗi P (như Brute Force). Thuật toán KMP sẽ giữ nguyên biến chạy của chuỗi T và dịch chuyển biến chạy của chuỗi P về vị trí phù hợp để giảm số lần so sánh.

Thuật toán Knuth–Morris–Pratt



Bản chất của việc dịch chuyển j về vị trí thích hợp là tận dụng kết quả của việc tìm kiếm lúc trước. Vì thế chúng ta lập một **mảng tiền tố dài nhất** của P (prefix) với prefix[i] là độ dài chuỗi con dài nhất bắt đầu từ 0, kết thúc tại i và trùng với tiền tố của P. Mảng này dùng để quay lại vị trí thích hợp mà không cần phải so sánh lại từ đầu.



Các giai đoạn xử lý

1. Giai đoạn tiền xử lý: Tạo ra mảng prefix cho chuỗi P.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	3	4

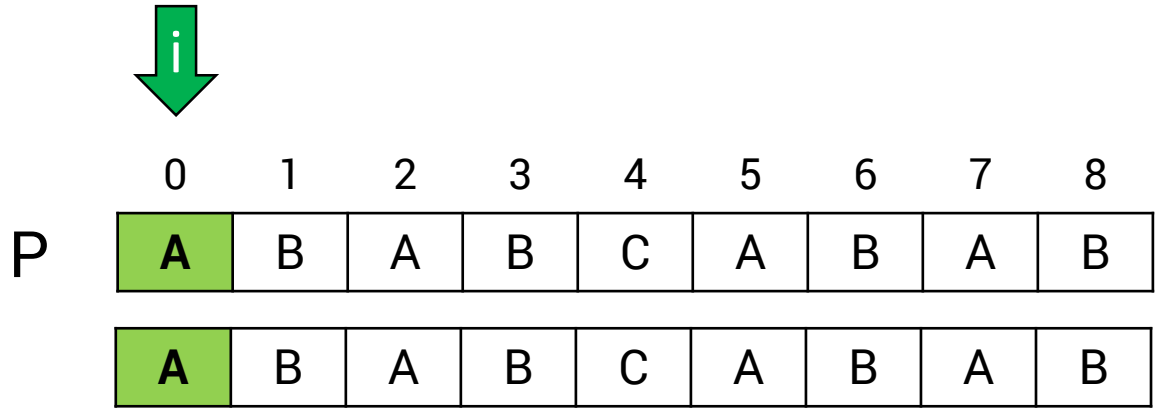
2. Giai đoạn tìm kiếm: So sánh các ký tự chuỗi P với chuỗi T, việc dịch chuyển tìm kiếm sẽ dựa vào mảng prefix của giai đoạn tiền xử lý.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	A	B	A	B	A	A	B	A	C	D	A	B	A	B	C	A	B	A	B

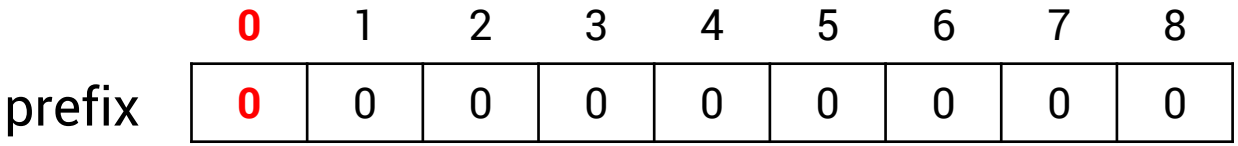
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
P	A	B	A	B	C	A	B	A	B

1. GIAI ĐOẠN TIỀN XỬ LÝ

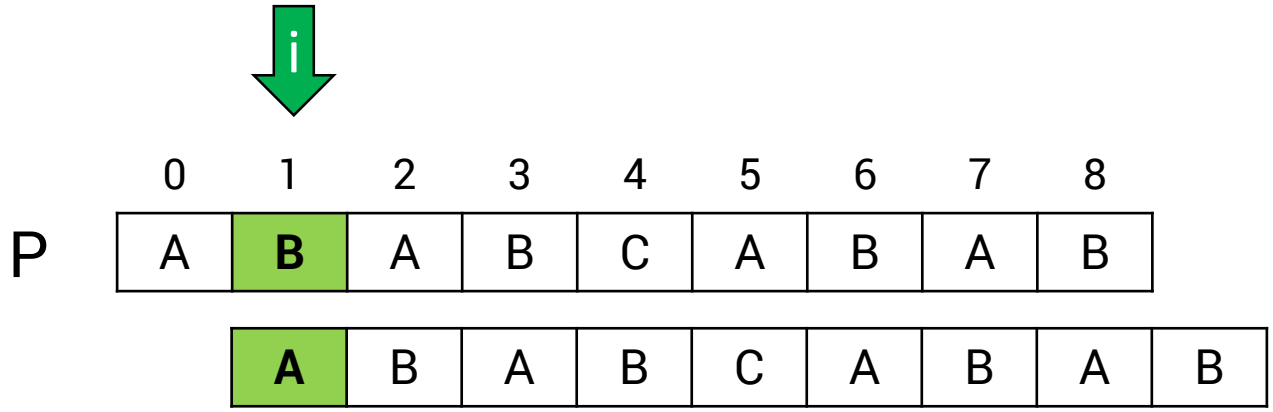
Bước 0: Tính prefix lần 0 (i=0)



Phần tử đầu tiên không có tiền tố trước đó. Giá trị đầu tiên của mảng prefix ➔ prefix[0] = 0.



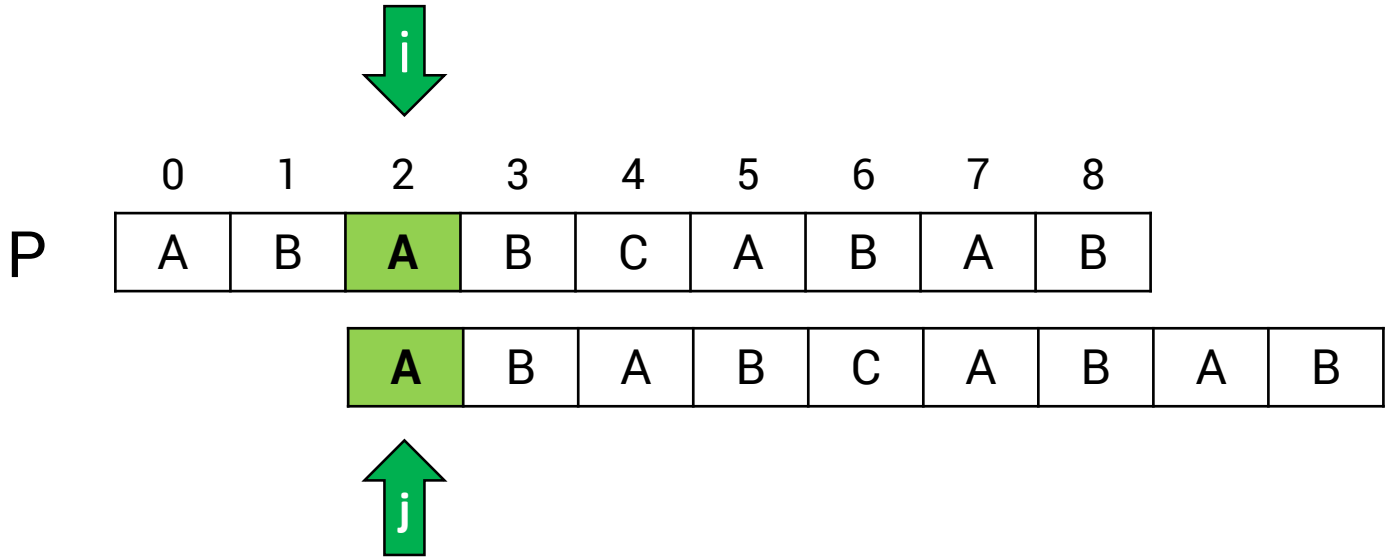
Bước 1: Tính prefix lần 1 (i=1)



Phần tử có index = 1, không có tiền tố trước đó (vì B khác A). Giá trị của mảng prefix ➔ prefix[1] = 0.

	0	1	2	3	4	5	6	7	8
prefix	0	0	0	0	0	0	0	0	0

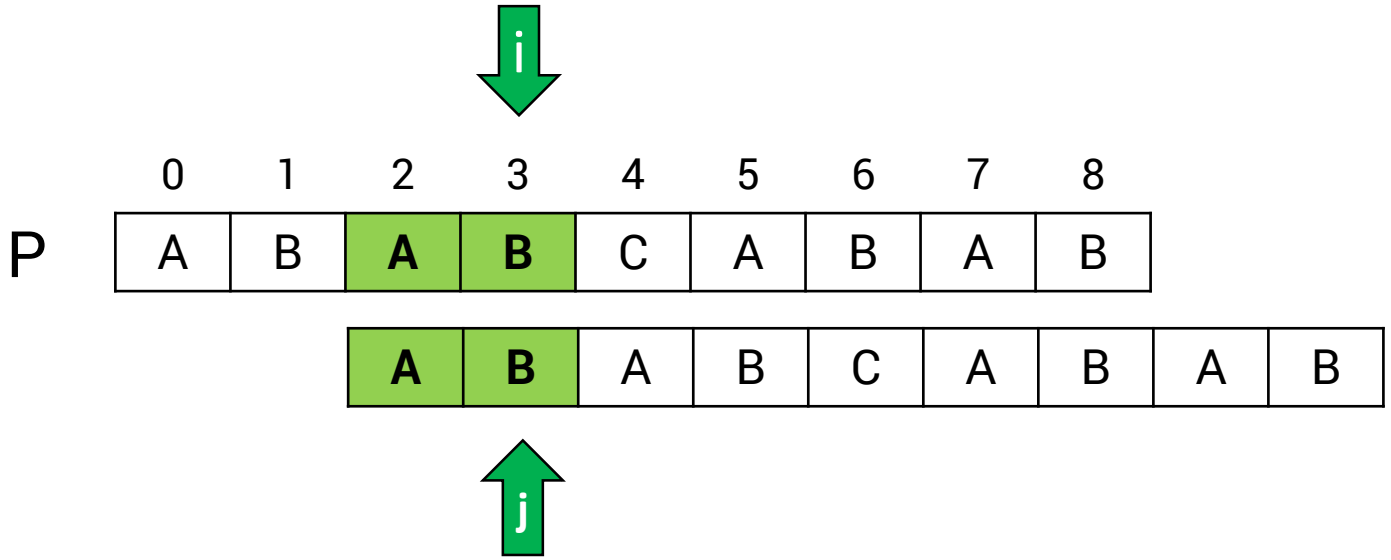
Bước 2: Tính prefix lần 2 (i=2)



Phần tử có index = 2, có 1 tiền tố trước đó (vì A giống A). Giá trị của mảng prefix ➔ prefix[2] = 1.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	0	0	0	0	0	0

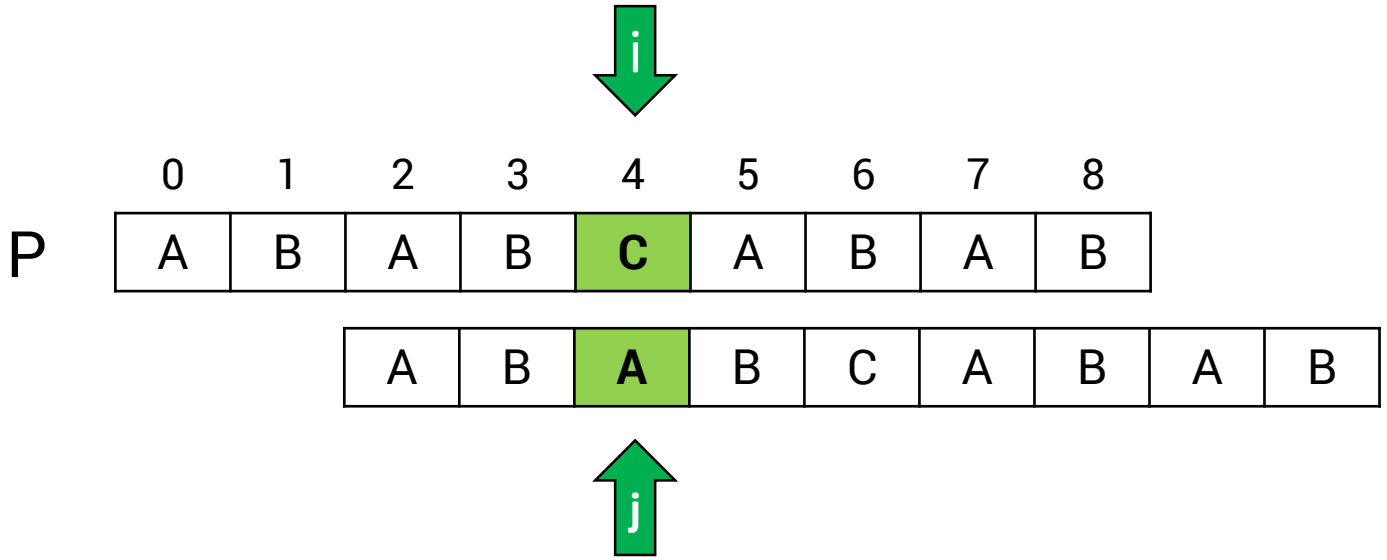
Bước 3: Tính prefix lần 3 (i=3)



Phần tử có index = 3, có 2 tiền tố trước đó (vì B giống B). Giá trị của mảng prefix ➔ prefix[3] = 2.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	0	0	0	0

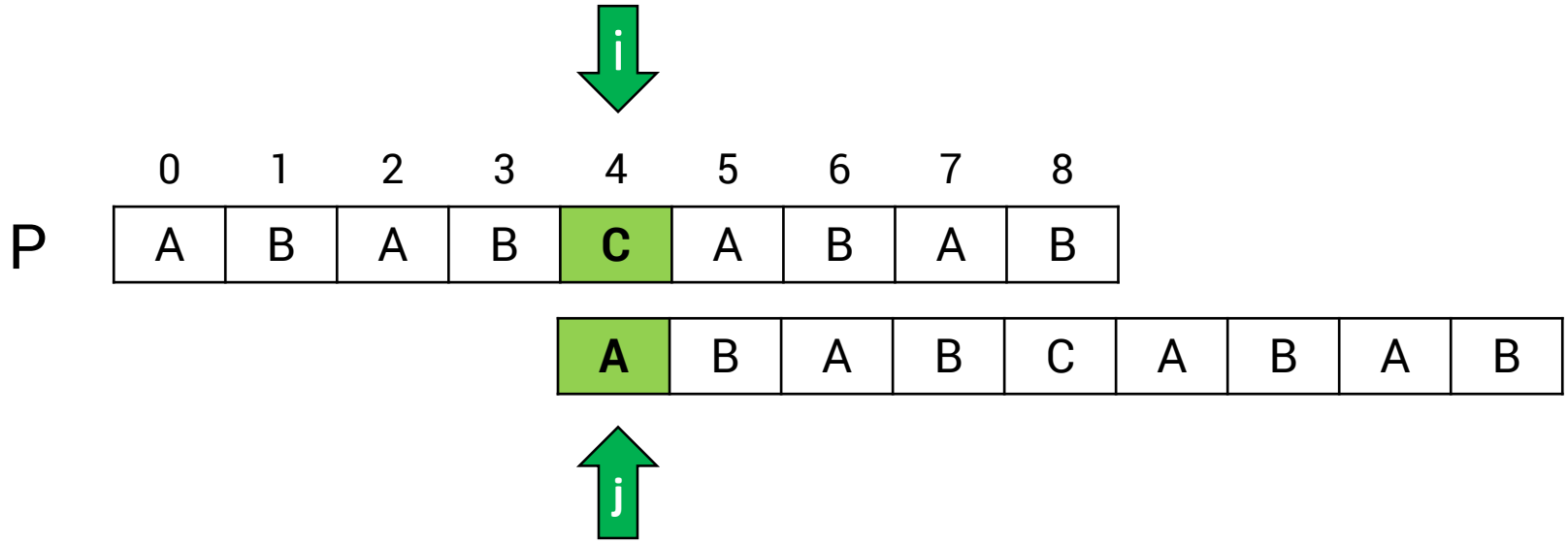
Bước 4: Tính prefix lần 4 (i=4)



Phần tử có index = 4, không có tiền tố trước đó (vì C khác A). Giá trị của mảng prefix ➔ prefix[4] = 0.

	0	1	2	3	**4**	5	6	7	8
prefix	0	0	1	2	**0**	0	0	0	0

Bước 4: Tính prefix lần 4 tiếp theo (i=4)

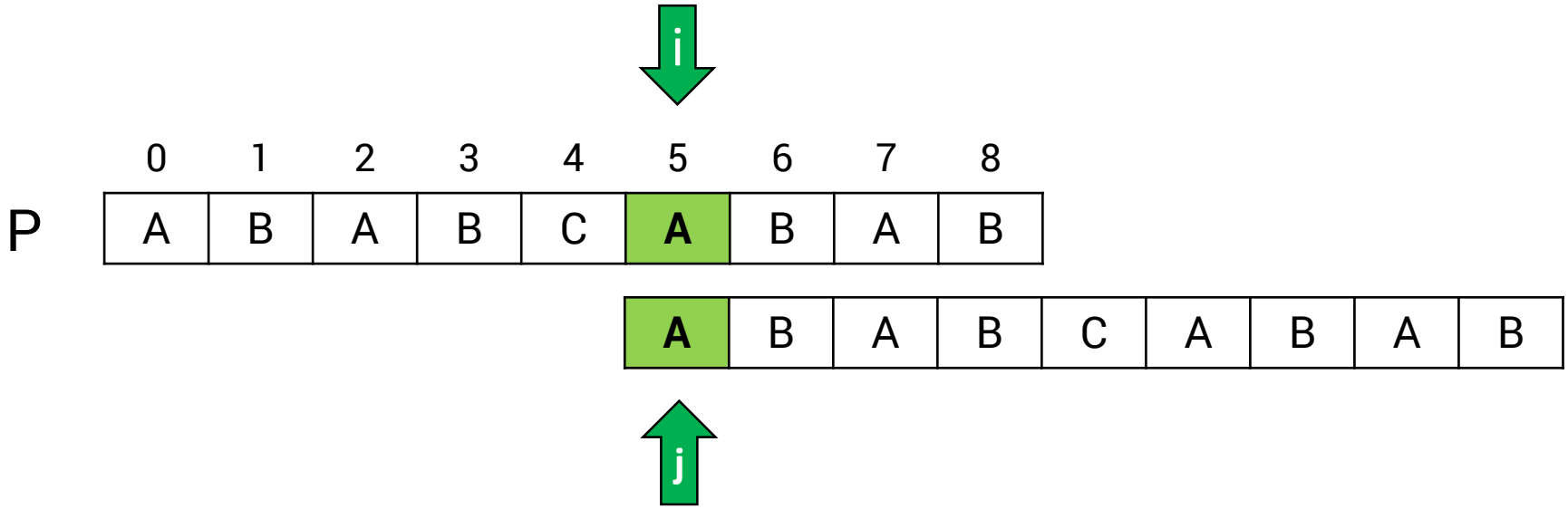


Xét lần 2, phần tử có index = 4, không tiền tố trước đó (vì C khác A).

Giá trị của mảng prefix ➔ prefix[4] = 0.

	0	1	2	3	**4**	5	6	7	8
prefix	0	0	1	2	**0**	0	0	0	0

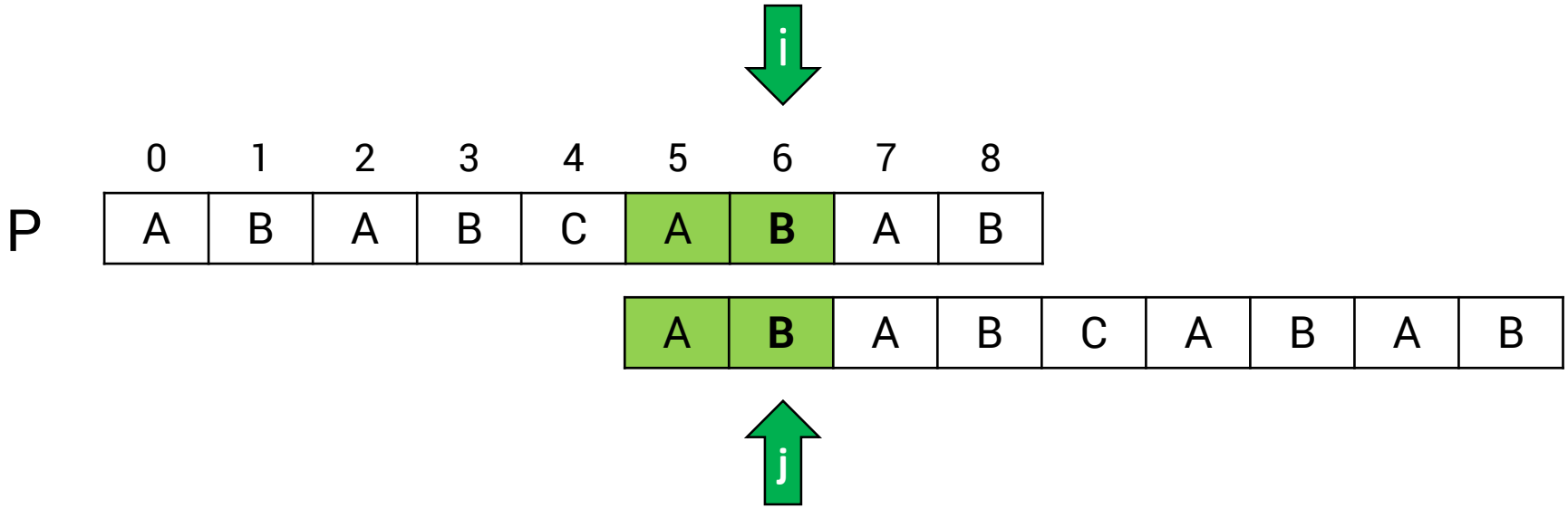
Bước 5: Tính prefix lần 5 (i=5)



Phần tử có index = 5, có 1 tiền tố trước đó (vì A giống A). Giá trị của mảng prefix ➔ prefix[5] = 1.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	0	0	0

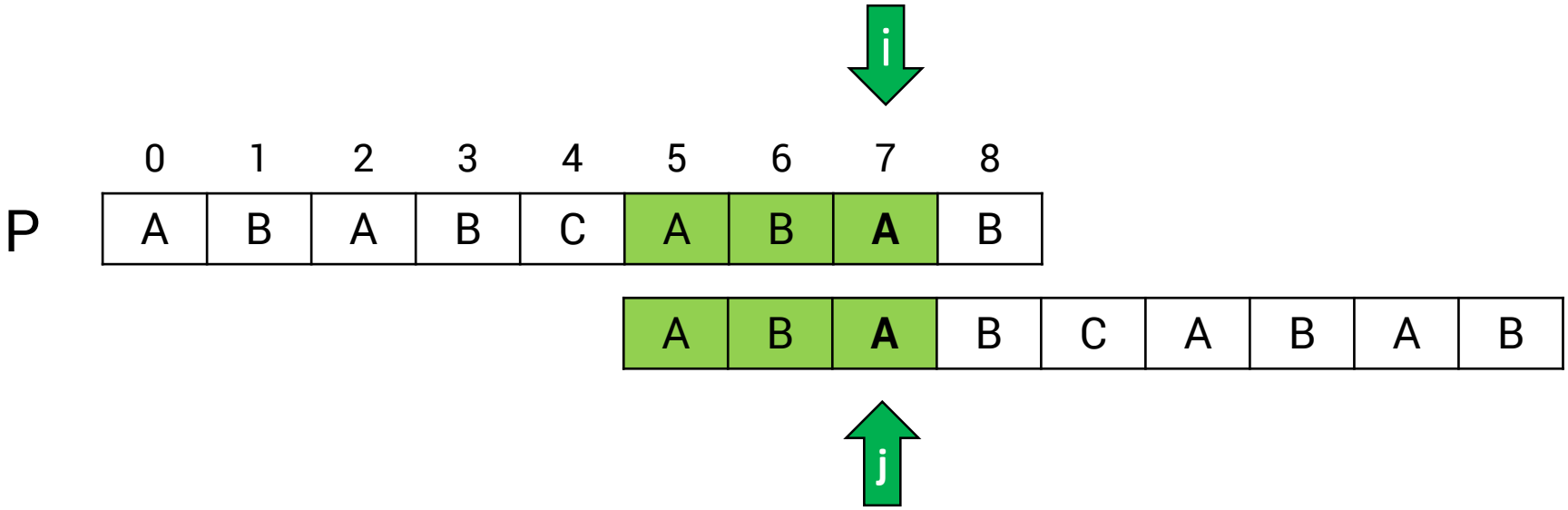
Bước 6: Tính prefix lần 6 (i=6)



Phần tử có index = 6, có 2 tiền tố trước đó (vì B giống B). Giá trị của mảng prefix ➔ prefix[6] = 2.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	0	0

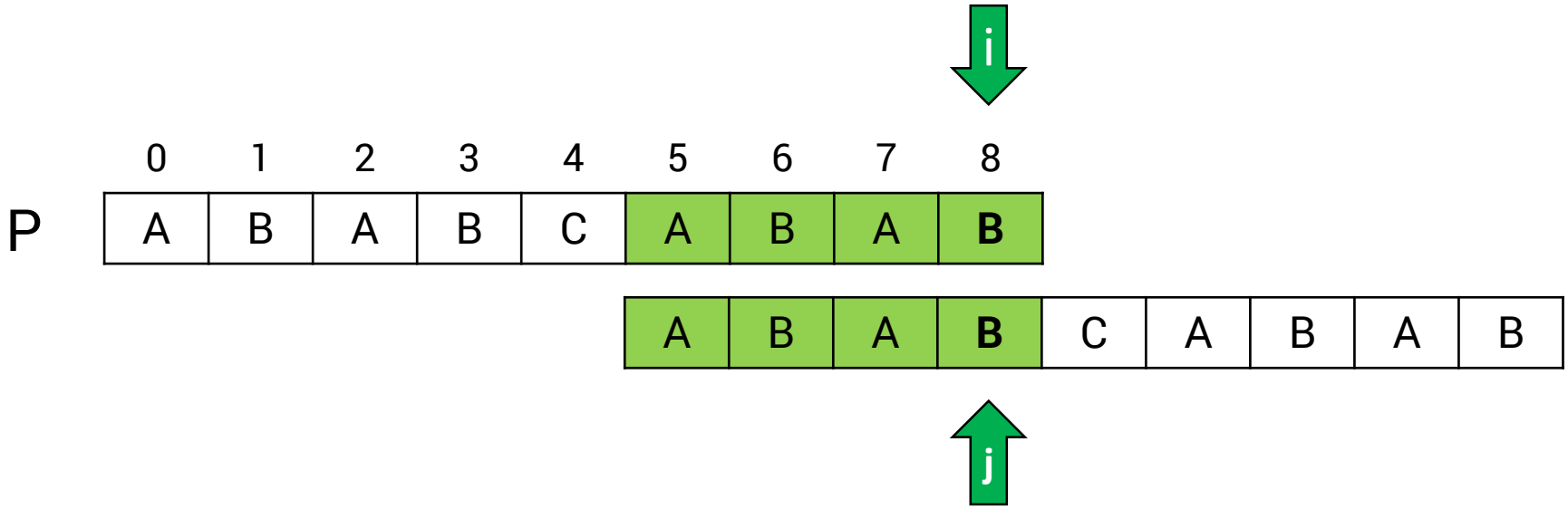
Bước 7: Tính prefix lần 7 (i=7)



Phần tử có index = 7, có 3 tiền tố trước đó (vì A giống A). Giá trị của mảng prefix ➔ prefix[7] = 3.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	**3**	0

Bước 8: Tính prefix lần 8 (i=8)



Phần tử có index = 8, có 4 tiền tố trước đó (vì B giống B). Giá trị của mảng prefix ➔ prefix[8] = 4.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	3	4

Kết quả thực hiện

i = 9 (độ dài của chuỗi P) → dừng thuật toán.

	0	1	2	3	4	5	6	7	8
P	A	B	A	B	C	A	B	A	B

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	3	4

Time Complexity: O(M)

- M là độ dài của chuỗi mẫu P.

Bài tập luyện tập

Bài tập 1:

	0	1	2	3	4	5	6	7	8
P	A	A	B	A	A	B	A	A	A

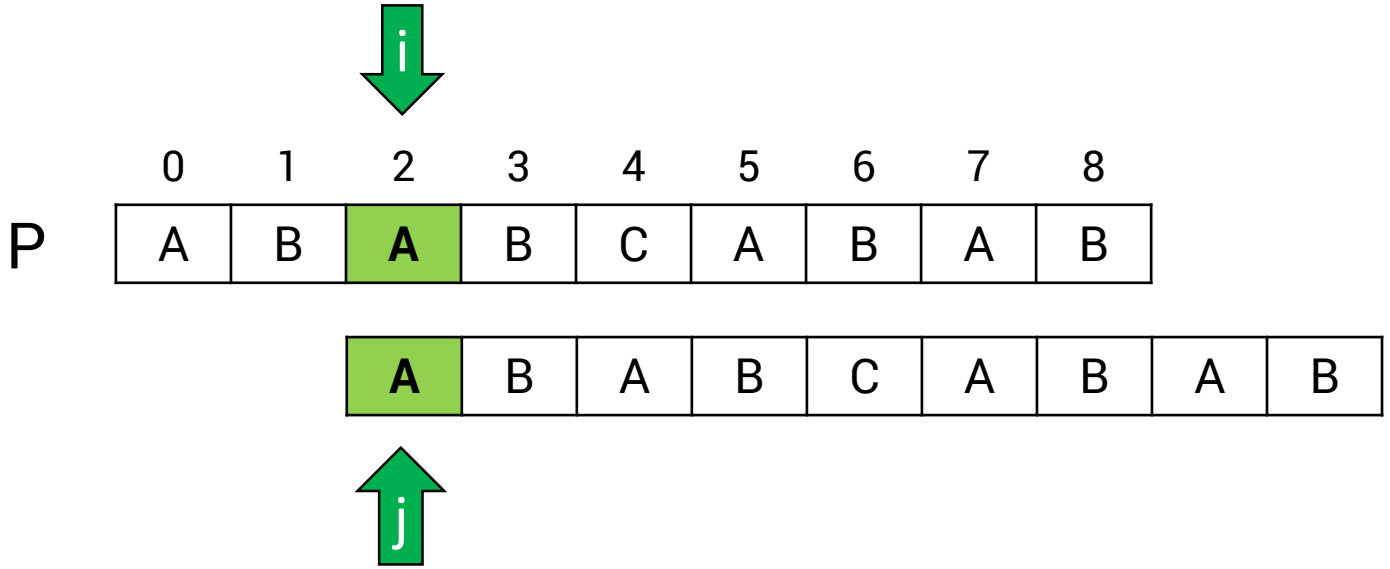
prefix

Bài tập 2:

	0	1	2	3	4	5	6	7
P	A	B	C	D	A	B	C	A

prefix

Cách tính mảng prefix



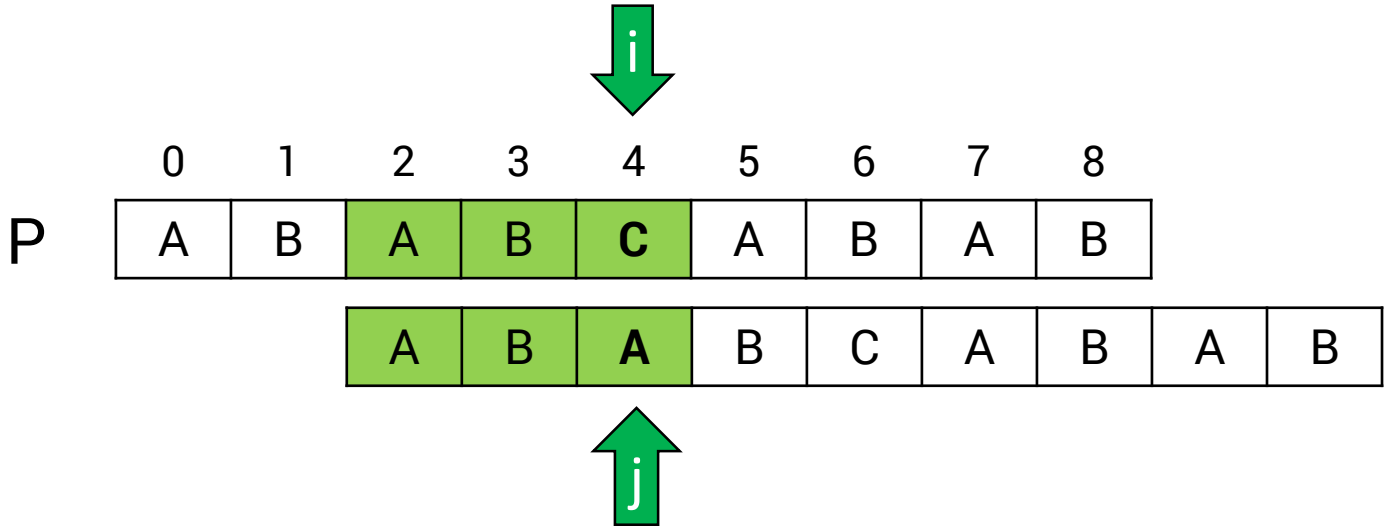
Nếu P[i] **giống** P[j] *(tồn tại tiền tố của ký tự đang xét)*

- **Tăng j lên 1 đơn vị, prefix[i] = j, tăng i lên 1 đơn vị (1)**

➔ j = 1, prefix[2] = 1, i = 3.

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	0	0	0	0	0	0

Cách tính mảng prefix



Nếu P[i] **khác** P[j] (không tồn tại tiền tố)

- j khác 0: *ta dời j về vị trí phù hợp với i và vẫn đảm bảo trùng với tiền tố của P* ➔ **Dời j về vị trí prefix[j-1] (2)**
- j bằng 0: *không thể dời được nữa, nghĩa là không có tiền tố nào có thể giữ* ➔ **prefix[i] = 0, tăng i lên 1 đơn vị (3)**

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	0	0	0	0

➔ **j = prefix[j-1] = prefix[1] = 0**

Source Code KMP Preprocess

```cpp
void KMPpreprocess(const string& p, vector<int>& prefix)
{
    prefix[0] = 0;
    int m = p.length();
    int j = 0;
    int i = 1;
    while (i < m)
    {
        if (p[i] == p[j])
        {
            j++;
            prefix[i] = j;
            i++;
        }
```

Source Code KMP Preprocess

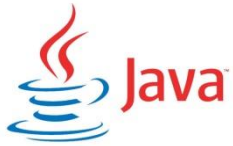
```
15.             else
16.             {
17.                 if (j != 0)
18.                     j = prefix[j - 1];
19.                 else
20.                 {
21.                     prefix[i] = 0;
22.                     i++;
23.                 }
24.             }
25.         }
26.     }
```

Source Code KMP Preprocess

```
1.  def KMPpreprocess(p, prefix):
2.      prefix[0] = 0
3.      m = len(p)
4.      j = 0
5.      i = 1
6.      while i < m:
7.          if p[i] == p[j]:
8.              j += 1
9.              prefix[i] = j
10.             i += 1
11.         else:
12.             if j != 0:
13.                 j = prefix[j - 1]
14.             else:
15.                 prefix[i] = 0
16.                 i += 1
```

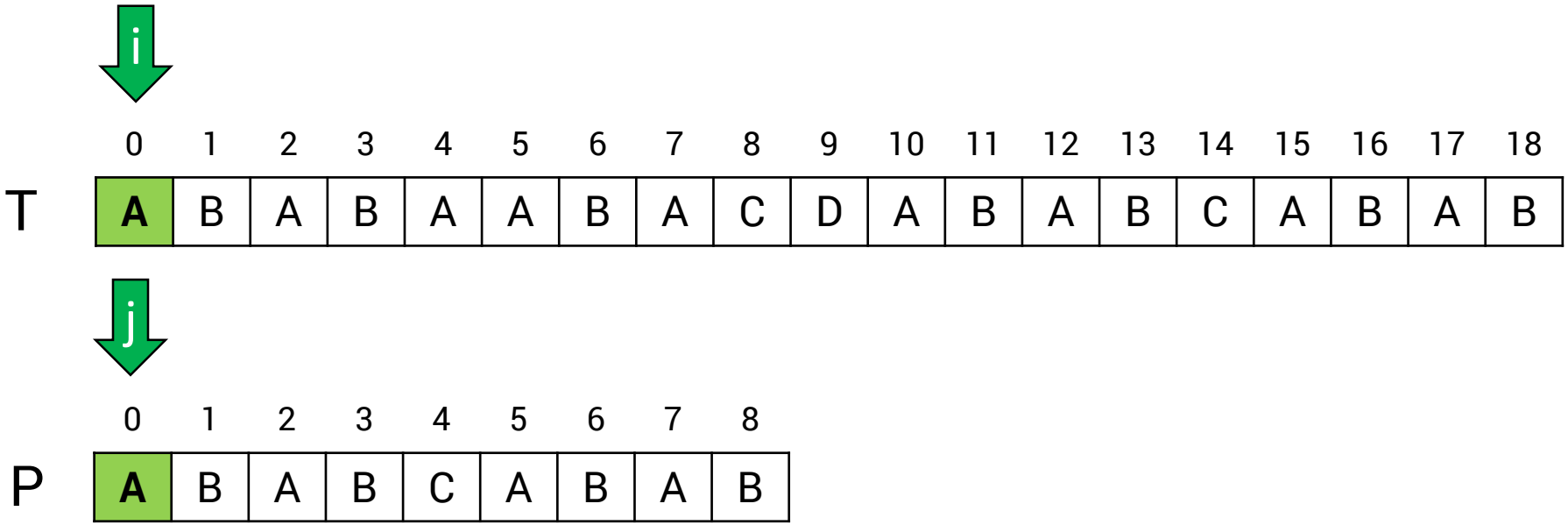
Source Code KMP Preprocess

```java
public class Main {
    private static void KMPpreprocess(String p, int[] prefix) {
        prefix[0] = 0;
        int m = p.length();
        int j = 0;
        int i = 1;
        while (i < m) {
            if (p.charAt(i) == p.charAt(j)) {
                j++;
                prefix[i] = j;
                i++;
            }
            else {
                if (j != 0)
                    j = prefix[j - 1];
                else {
                    prefix[i] = 0;
                    i++;
                }
            }
        }
    }
```



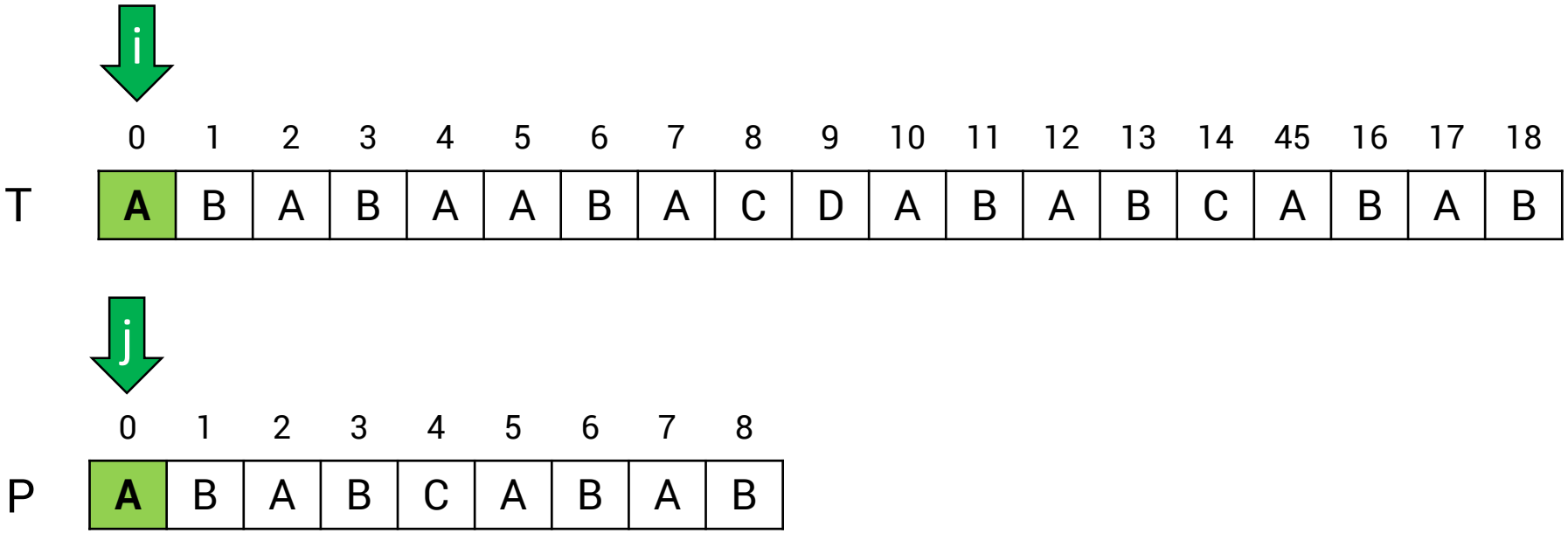
2. GIAI ĐOẠN TÌM KIẾM: SO SÁNH CHUỖI

Cách chạy so sánh chuỗi



- Nếu T[i] giống P[j]
  - *Tồn tại ký tự giống nhau 2 chuỗi* ➔ **tăng i và j lên 1 đơn vị (1)**
- Nếu T[i] khác P[j]
  - j khác 0: dời j về nhưng vẫn giữ đoạn tiền tố trùng khớp dài nhất để không phải so sánh lại ➔ **dời j về vị trí prefix[j-1] (2)**
  - j bằng 0: Chuỗi P không thể xuất hiện tại vị trí i➔ **tăng i lên 1 đơn vị (3)**

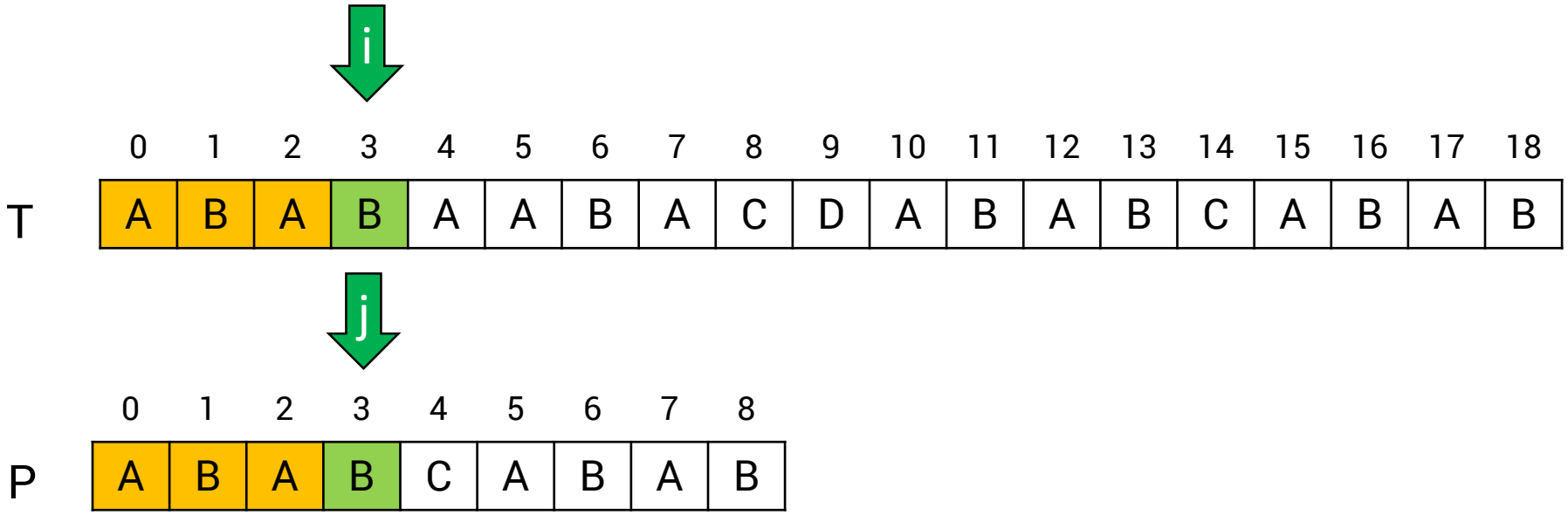
Bước 1: So sánh chuỗi lần 1 (i=0)



i	j	T[i]	P[j]	i+=1, j+=1 (1)	
0	0	A	A	i = 1	j = 1

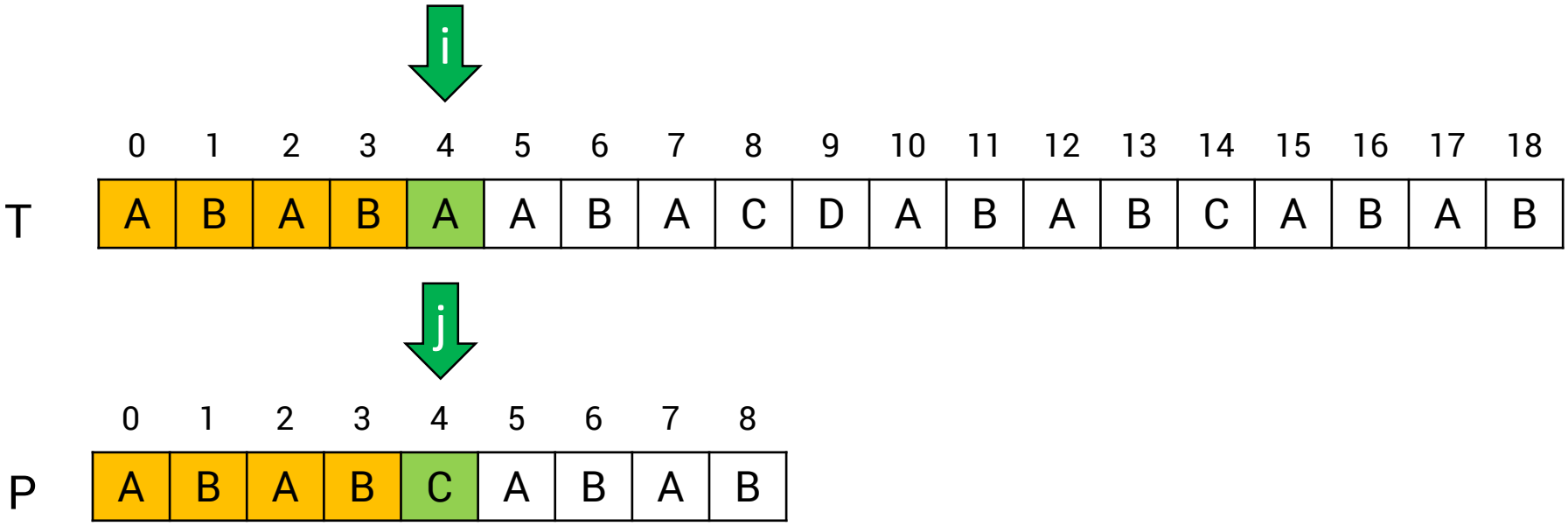
Bước 2, 3, 4: so sánh chuỗi (i=1, 2, 3)

Tương tự tăng i và j lên các giá trị 1, 2, 3: T[i] và P[j] đều giống nhau.



i	j	T[i]	P[j]	**i+=1, j+=1 (1)**	
1	1	B	B	i = 2	j = 2
2	2	A	A	i = 3	j = 3
3	3	B	B	i = 4	j = 4

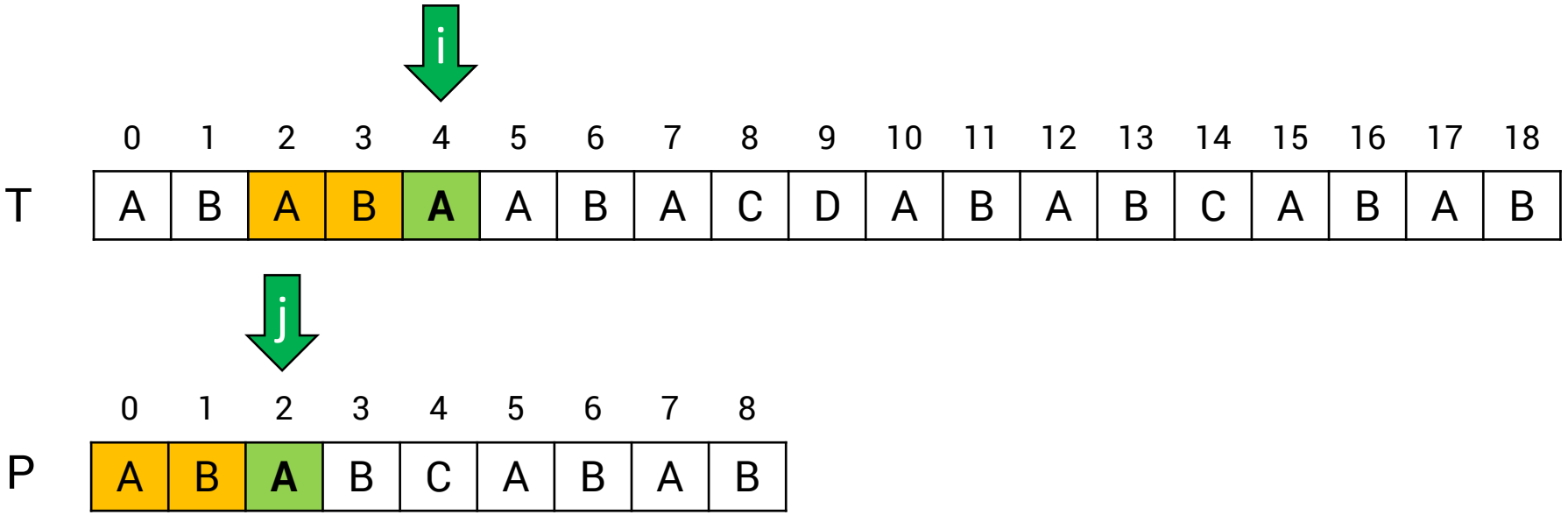
Bước 5: So sánh chuỗi lần 5 (i=4)



i	j	T[i]	P[j]	**j = prefix[j-1] (2)**
4	4	A	C	j = prefix[3] = 2

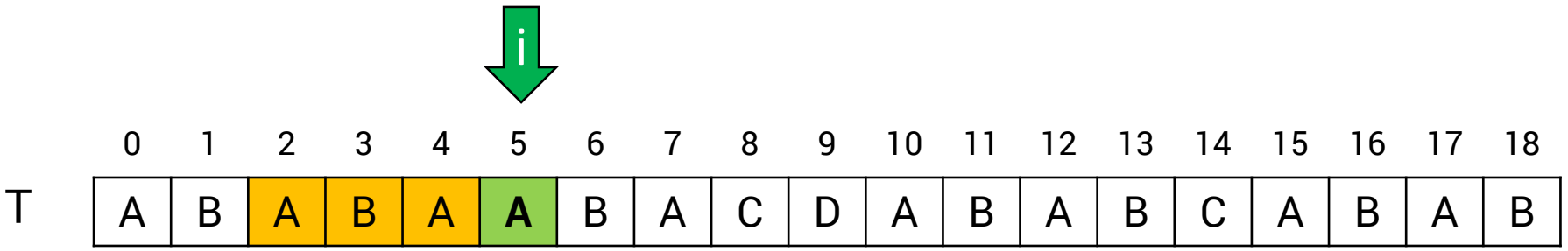
	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	3	4

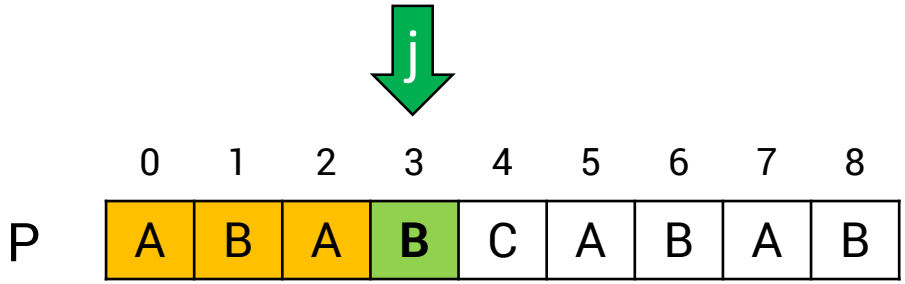
Bước 5: So sánh chuỗi lần 5 (i=4)



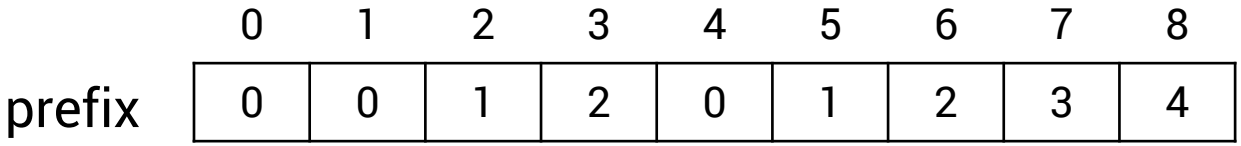
i	j	T[i]	P[j]	i+=1, j+=1 (1)	
4	2	B	B	i = 5	j = 3

Bước 6: So sánh chuỗi lần 6 (i=5)

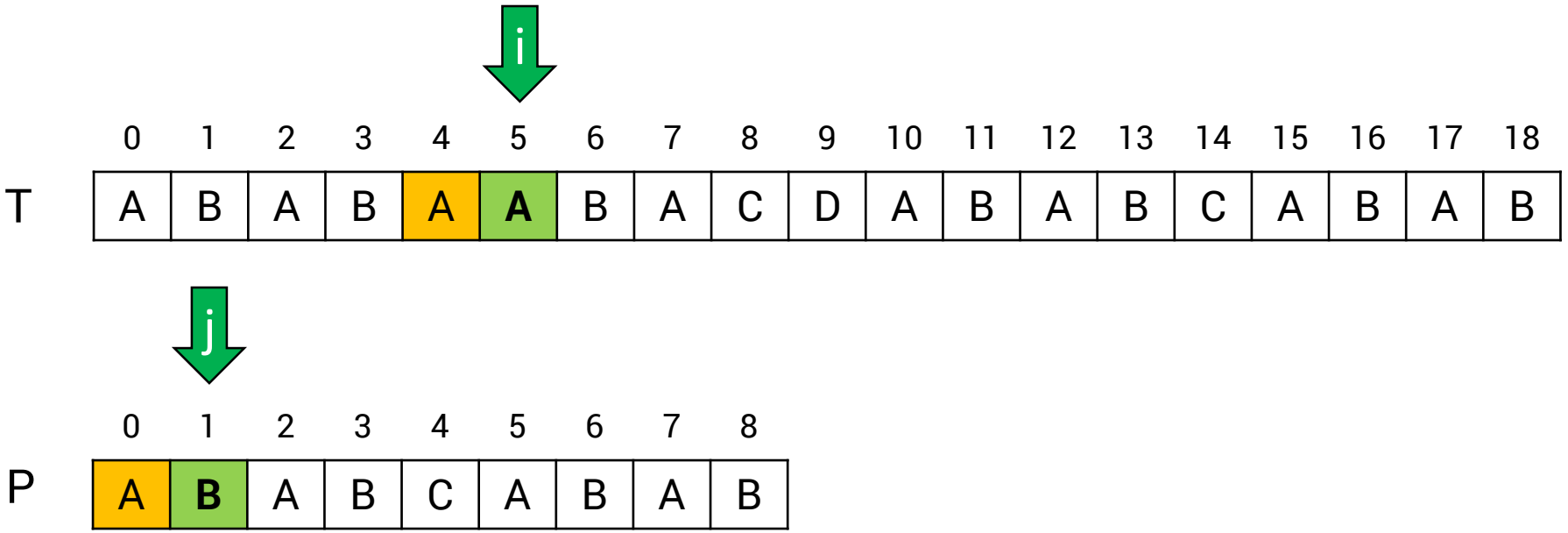




i	j	T[i]	P[j]	**j = prefix[j-1] (2)**
5	3	A	B	j = prefix[2] = 1



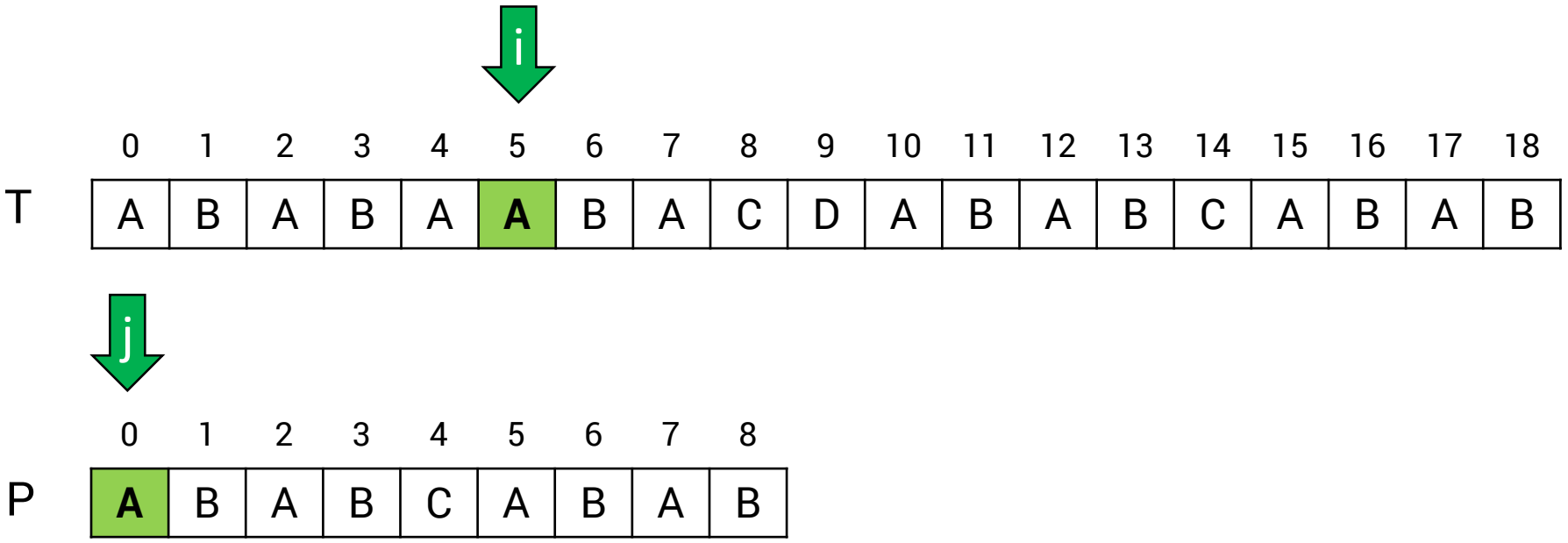
Bước 6: So sánh chuỗi lần 6 (i=5)



i	j	T[i]	P[j]	**j = prefix[j-1] (2)**
5	1	A	B	j = prefix[0] = 0

	0	1	2	3	4	5	6	7	8
prefix	0	0	1	2	0	1	2	3	4

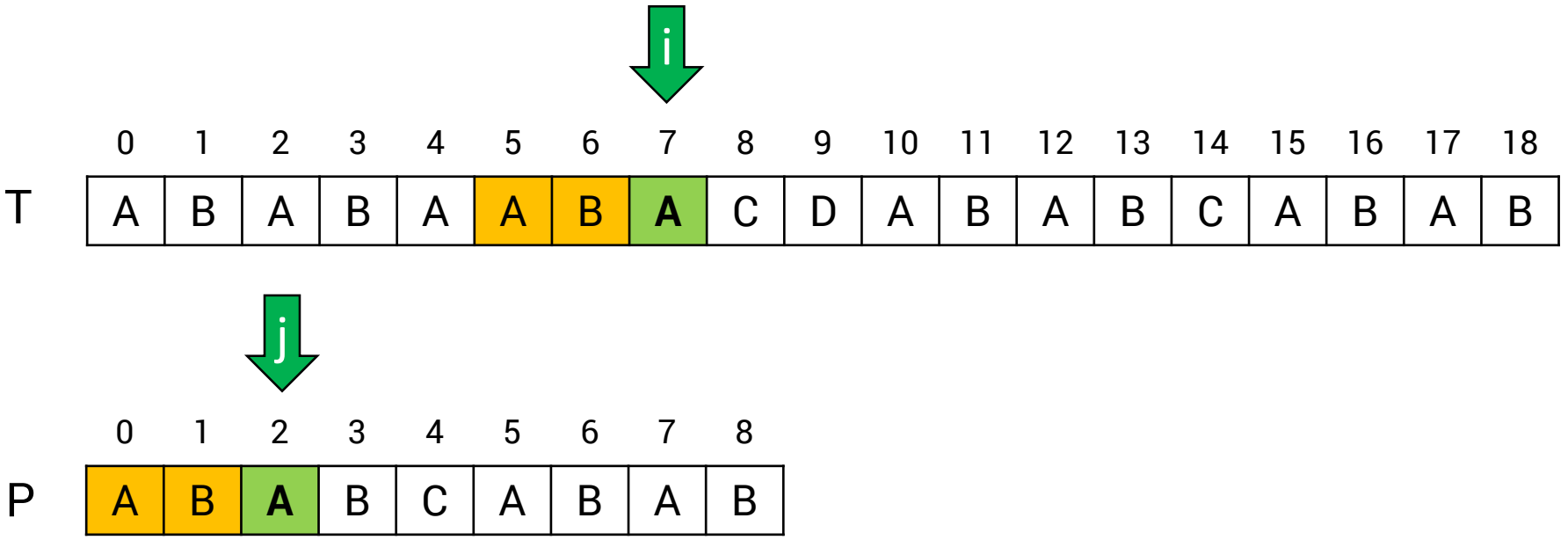
Bước 6: So sánh chuỗi lần 6 (i=5)



i	j	T[i]	P[j]	i+=1, j+=1 (1)	
5	0	A	A	i = 6	j = 1

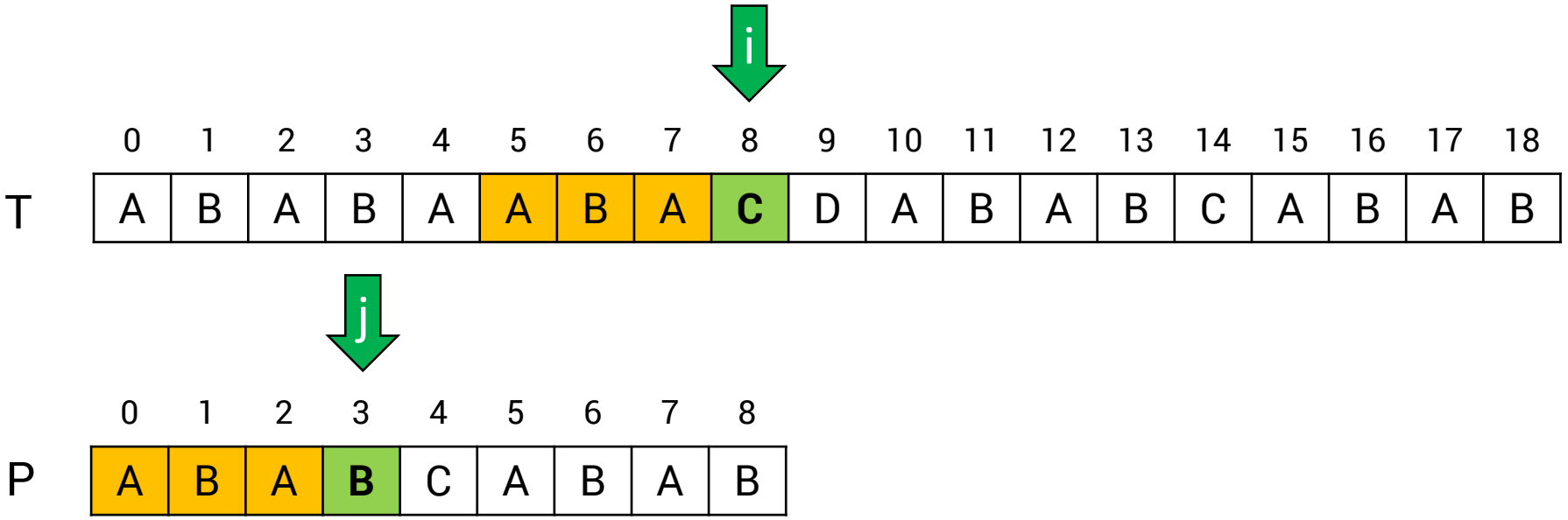
Bước 7, 8: So sánh chuỗi (i = 6, 7)

Tương tự tăng i (lên 6, 7) và j (lên 1, 2) các giá trị của T[i] và P[j] đều giống nhau.

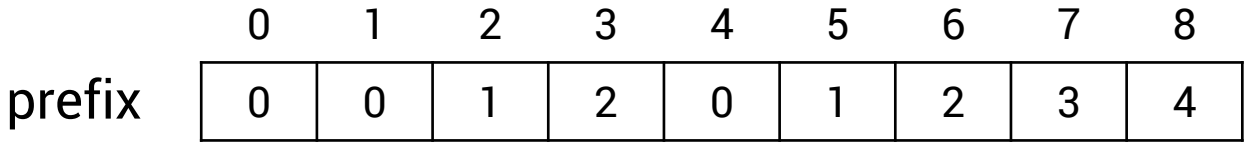


i	j	T[i]	P[j]	**i+=1, j+=1 (1)**	
6	1	B	B	i = 7	j = 2
7	2	A	A	i = 8	j = 3

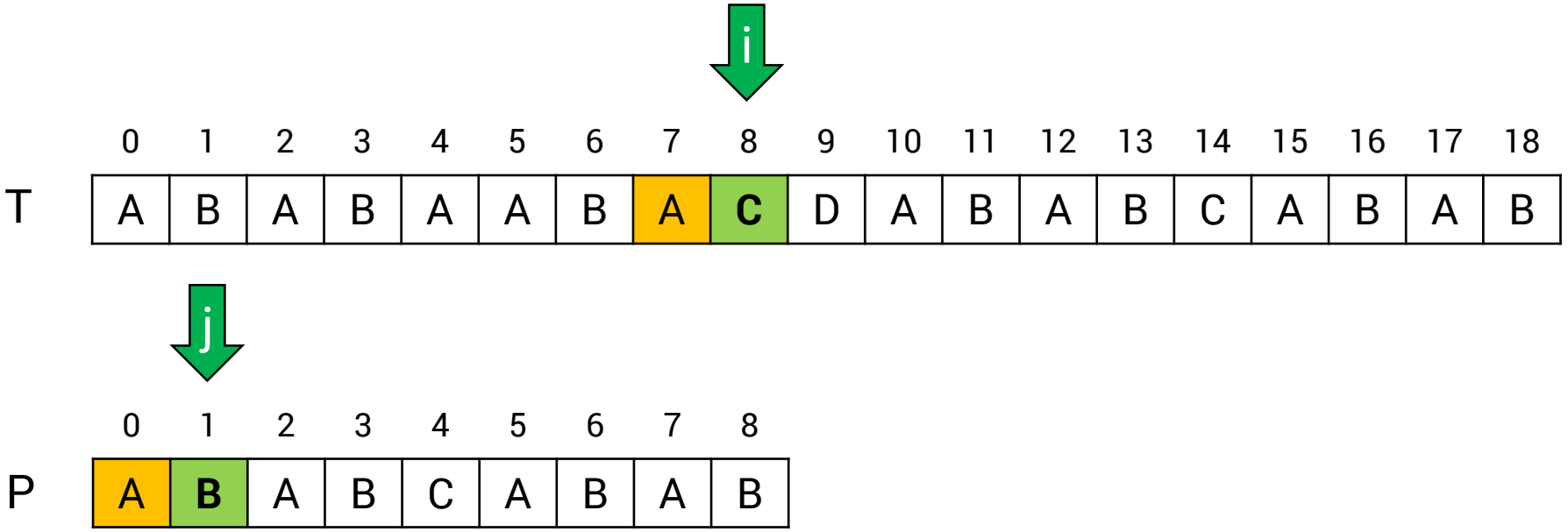
Bước 9: So sánh chuỗi lần 9 (i=8)



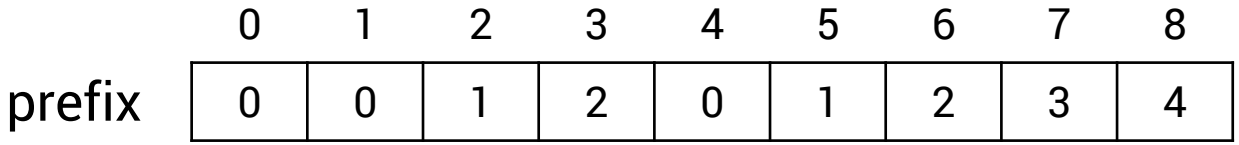
i	j	T[i]	P[j]	**j = prefix[j-1] (2)**
8	3	C	B	j = prefix[2] = 1



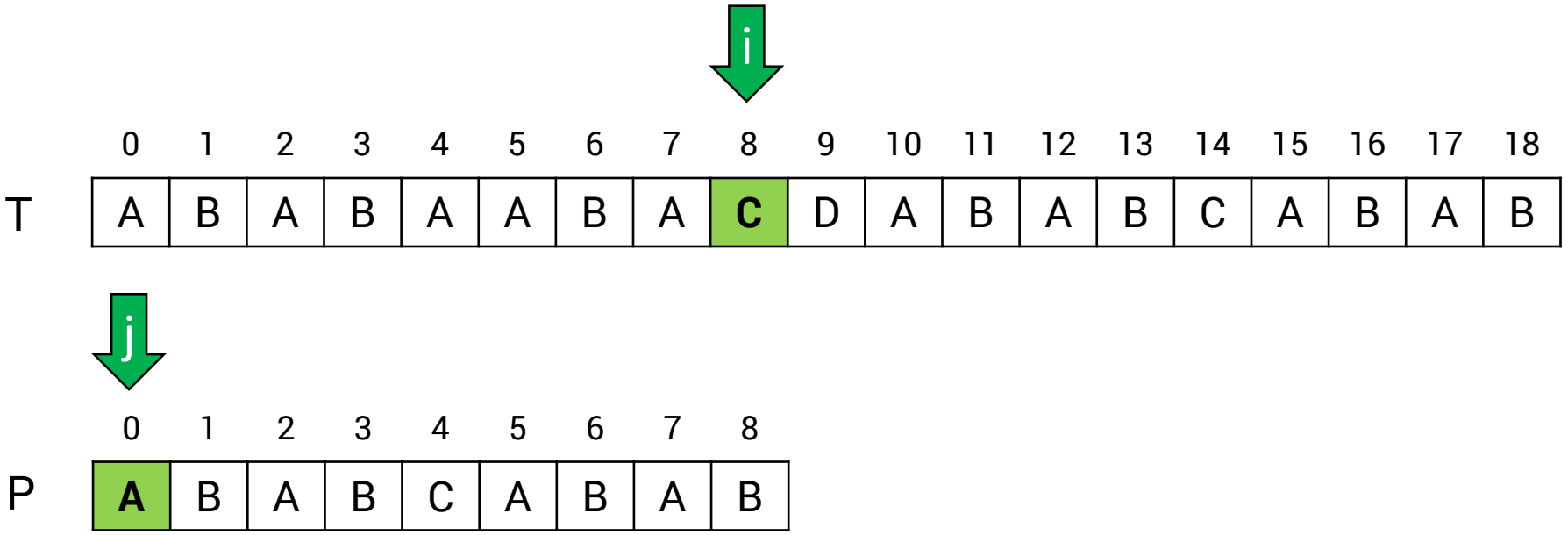
Bước 9: So sánh chuỗi lần 9 (i=8)



i	j	T[i]	P[j]	**j = prefix[j-1] (2)**
8	1	C	B	j = prefix[0] = 0

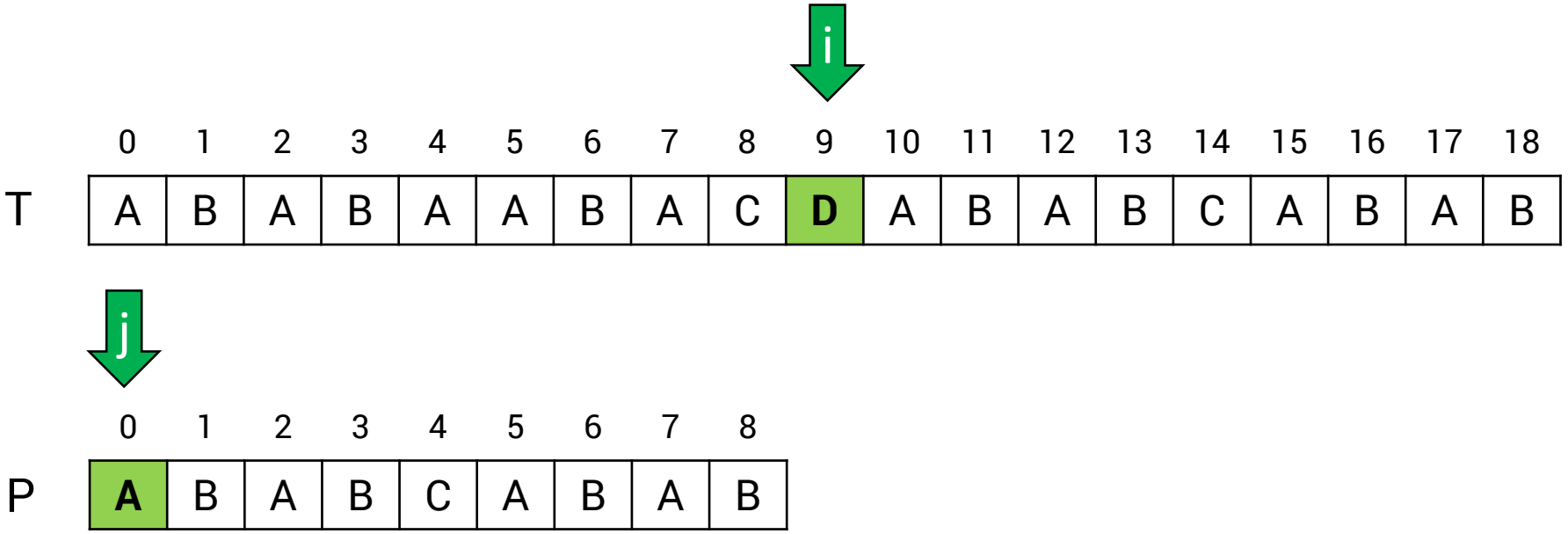


Bước 9: So sánh chuỗi lần 9 (i=8)



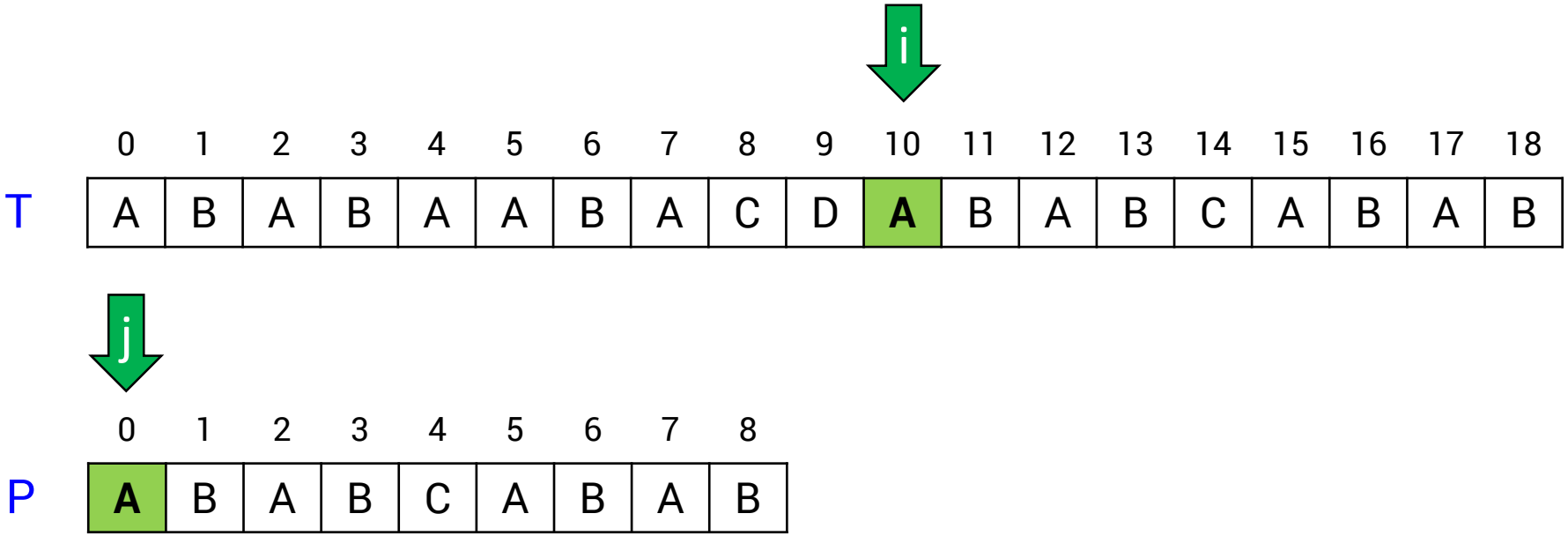
i	j	T[i]	P[j]	**i+=1 (3)**
8	0	C	A	i = 9

Bước 10: So sánh chuỗi lần 10 (i=9)



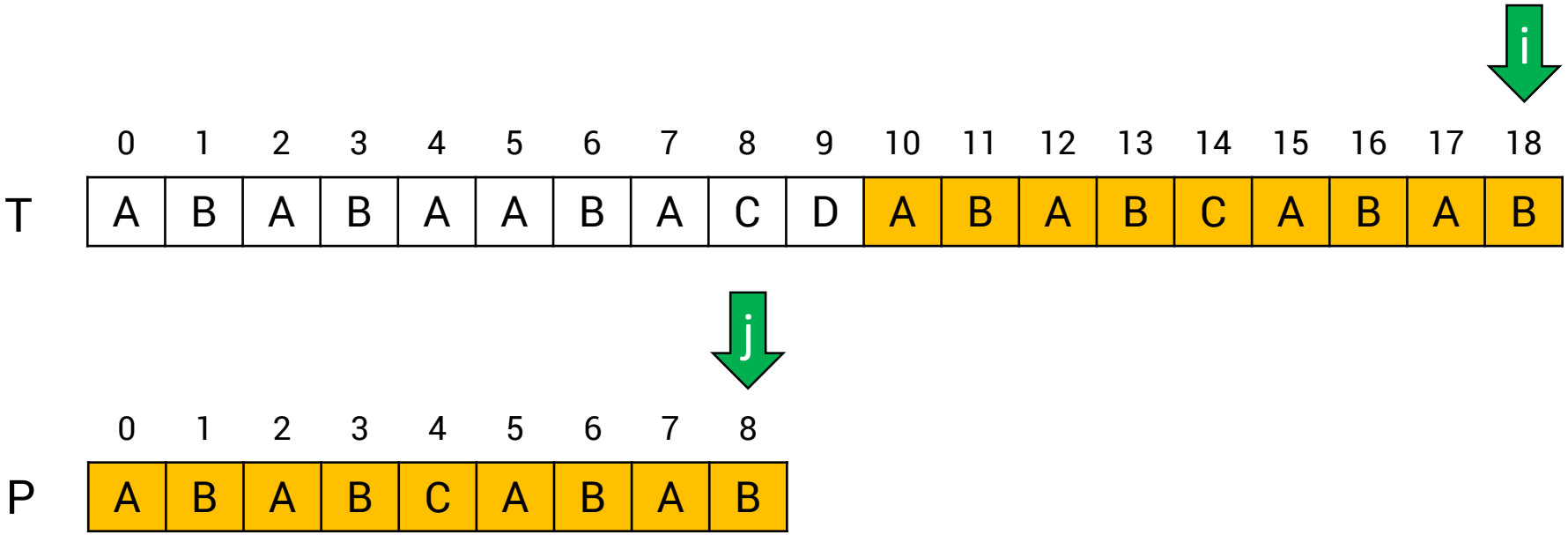
i	j	T[i]	P[j]	**i+=1 (3)**
9	0	D	A	i = 10

Bước 11: So sánh chuỗi lần 11(i=10)



i	j	T[i]	P[j]	i+=1, j+=1 (1)	
10	0	A	A	i = 11	j = 1

Bước 12 đến 19: tiếp tục so sánh chuỗi



i	j	T[i]	P[j]	Kết quả (i – j)
18	8	B	B	10

➔ Vị trí xuất hiện của chuỗi P trong chuỗi T:

10

Source Code KMP Search

```
1.  void KMPsearch(const string& t, const string& p, const vector<int>& prefix)
2.  {
3.      int n = t.length();
4.      int m = p.length();
5.      int i = 0, j = 0;
6.      while (i < n)
7.      {
8.          if (p[j] == t[i])
9.          {
10.             j++;
11.             i++;
12.         }
13.         if (j == m)
14.         {
15.             cout << "Found pattern at index: " << (i - j) << "\n";
16.             j = prefix[j - 1];
17.         }
```

Source Code KMP Search

```
18.             else if (i < n && p[j] != t[i])
19.             {
20.                 if (j != 0)
21.                 {
22.                     j = prefix[j - 1];
23.                 }
24.                 else
25.                 {
26.                     i = i + 1;
27.                 }
28.             }
29.         }
30.     }
```

Source Code KMP Search

```
31. int main()
32. {
33.     string t = "ABABAABACDABABCABAB";
34.     string p = "ABABCABAB";
35.     vector<int> prefix(p.length());
36.     KMPpreprocess(p, prefix);
37.     KMPsearch(t, p, prefix);
38.     return 0;
39. }
```

Source Code KMP Search

```
1.  def KMPsearch(t, p, prefix):
2.      n = len(t)
3.      m = len(p)
4.      i = j = 0
5.      while i < n:
6.          if p[j] == t[i]:
7.              i += 1
8.              j += 1
9.          if j == m:
10.             print('Found pattern at index:', i - j);
11.             j = prefix[j - 1]
12.         elif i < n and p[j] != t[i]:
13.             if j != 0:
14.                 j = prefix[j - 1]
15.             else:
16.                 i += 1
```

Source Code KMP Search

```
17. if __name__ == "__main__":
18.     t = "ABABDABACDABABCABAB"
19.     p = "ABABCABAB"
20.     prefix = [0] * len(p)
21.     KMPpreprocess(p, prefix)
22.     KMPsearch(t, p, prefix)
```

Source Code KMP Search

```
1.      private static void KMPsearch(String t, String p, int[] prefix) {
2.          int n = t.length();
3.          int m = p.length();
4.          int i = 0, j = 0;
5.          while (i < n) {
6.              if (p.charAt(j) == t.charAt(i)) {
7.                  j++;
8.                  i++;
9.              }
10.             if (j == m) {
11.                 System.out.printf("Found pattern at index: %d\n", i - j);
12.                 j = prefix[j - 1];
13.             }
```

Source Code KMP Search

```
14.                 else if (i < n && p.charAt(j) != t.charAt(i)) {
15.                     if (j != 0) {
16.                         j = prefix[j - 1];
17.                     }
18.                     else {
19.                         i = i + 1;
20.                     }
21.                 }
22.             }
23.         }
```

Source Code KMP Search

```
24.     public static void main(String[] args) {
25.         String t = "ABABDABACDABABCABAB";
26.         String p = "ABABCABAB";
27.         int[] prefix = new int[p.length()];
28.         KMPpreprocess(p, prefix);
29.         KMPsearch(t, p, prefix);
30.     }
31. }
```

Hỏi đáp