Pattern matching: part deux

Lecture 23

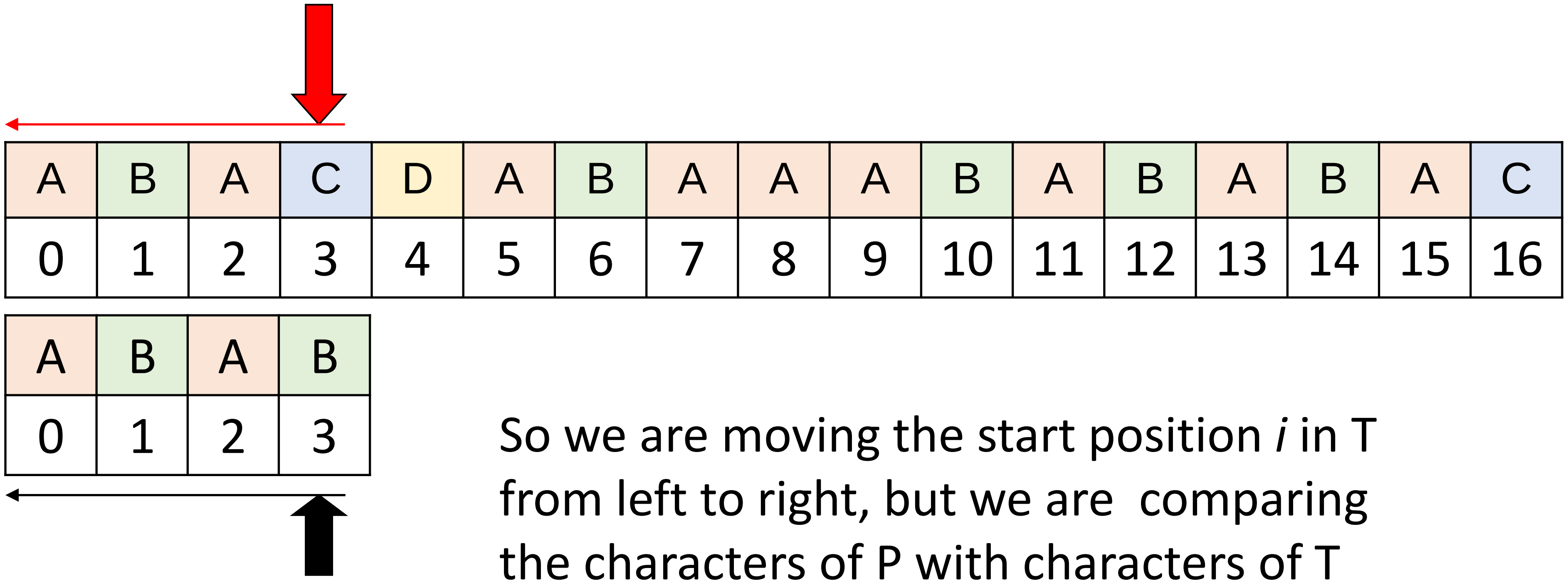
Shifting heuristics

- In the naive algorithm, we check the occurrence of pattern P in text T, each time aligning the start of P with the next character in T

- In the KMP algorithm we speed up the search by using some information about pattern P and the characters of T that were already aligned with P: This allows us to shift the start of P by more than one position in T

New idea

- As before, we first align the position 0 of the pattern with position 0 of the text
- However, we compare characters of P **starting from the last position of P**, position M - 1 (where M is the length of P)

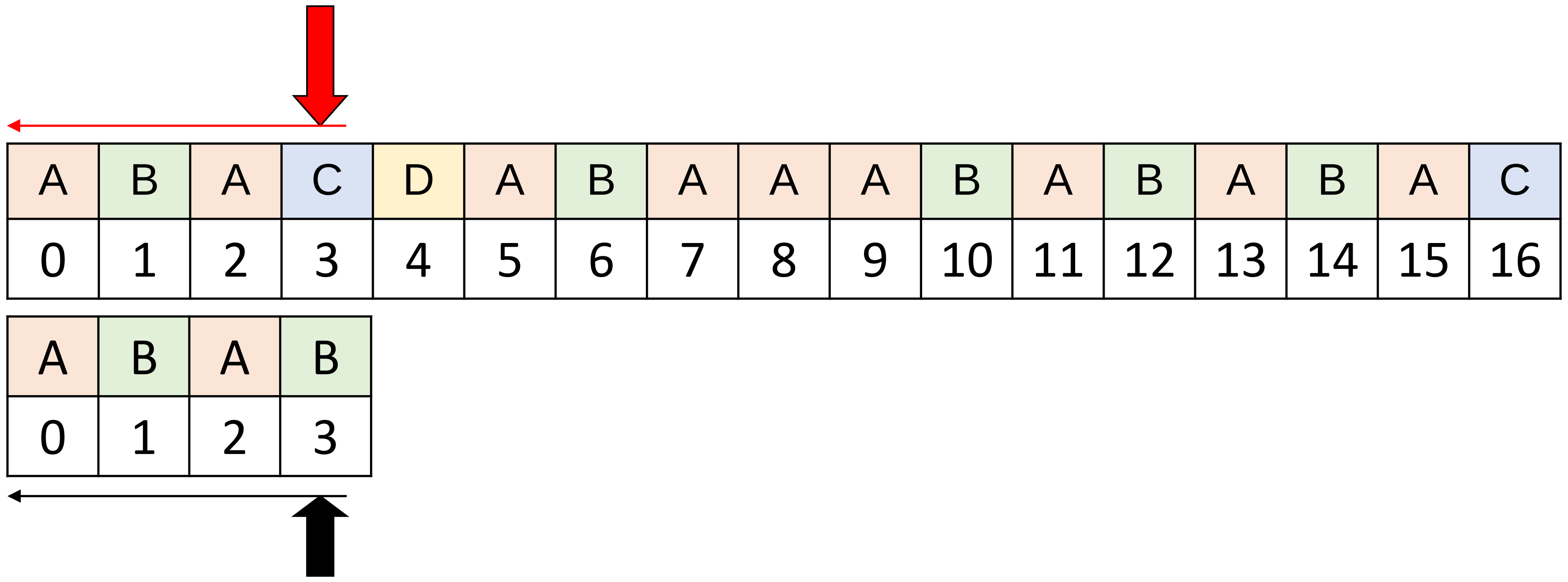


A	B	A	C	D	A	B	A	A	A	B	A	B	A	B	A	C
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

A	B	A	B
0	1	2	3

So we are moving the start position *i* in T from left to right, but we are comparing the characters of P with characters of T from right to left.

New idea

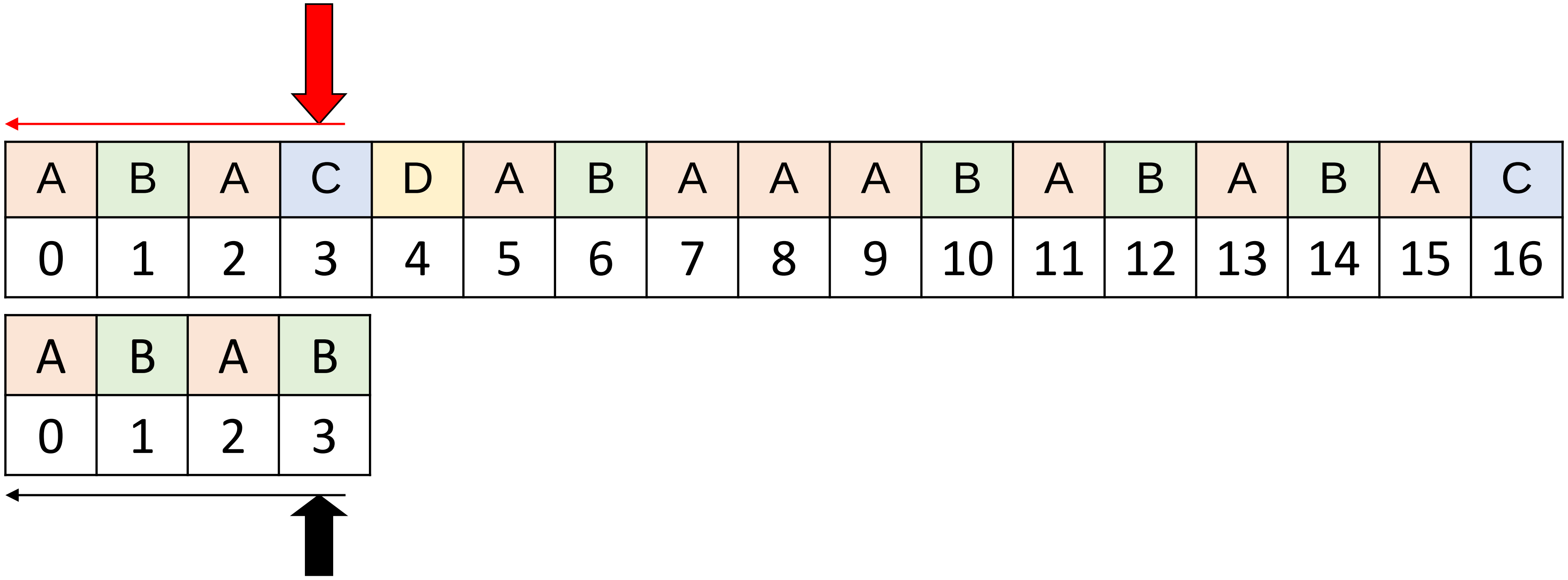


If we first compare C and B, we learn two things:

- C does not match B
- C does not appear anywhere in the pattern

Should we continue comparing T[2] and P[2]?

New idea



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

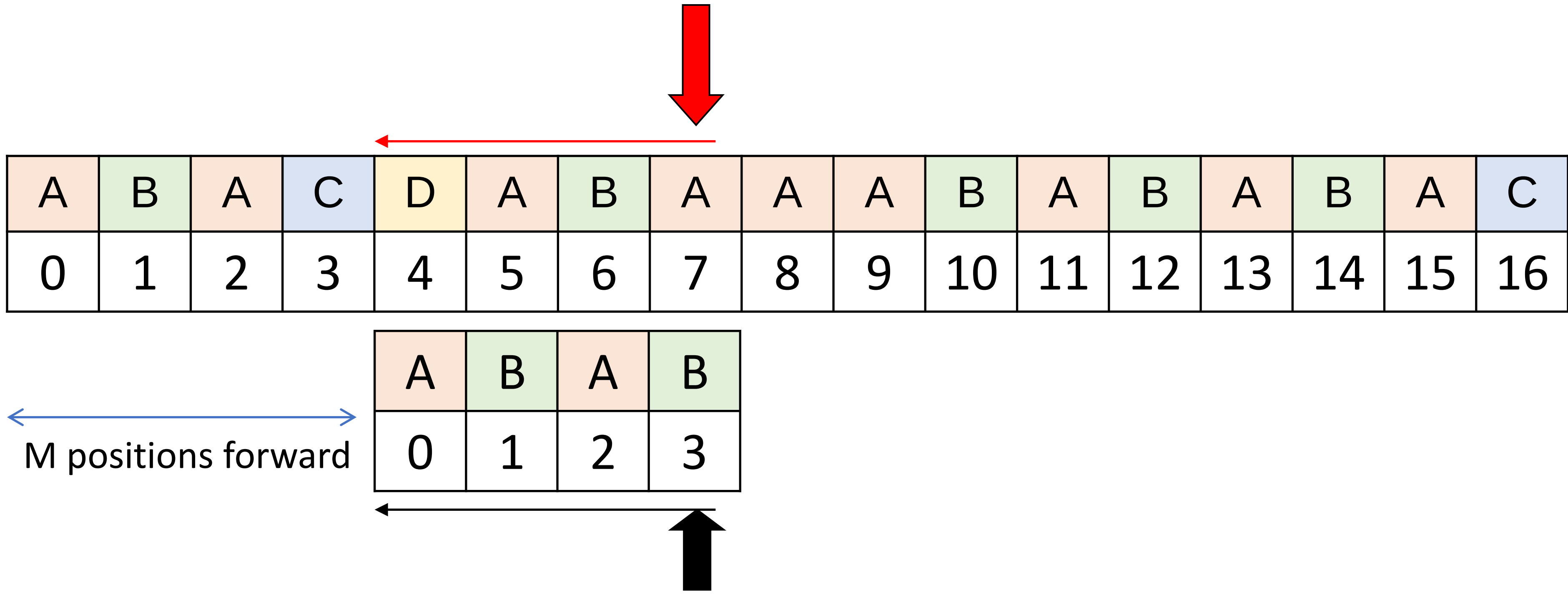
A	B	A	B
0	1	2	3

If we first compare C and B, we learn two things:

- C does not match B
- C does not appear anywhere in the pattern

- Where should we position the pattern for the next round of comparisons?
- Which positions in T and P will we compare next?

New idea



With one mismatch we shifted down the entire length of the pattern (M positions)!

- Some characters of T will be not compared at all!
- This could be a sublinear search! (< N comparisons)

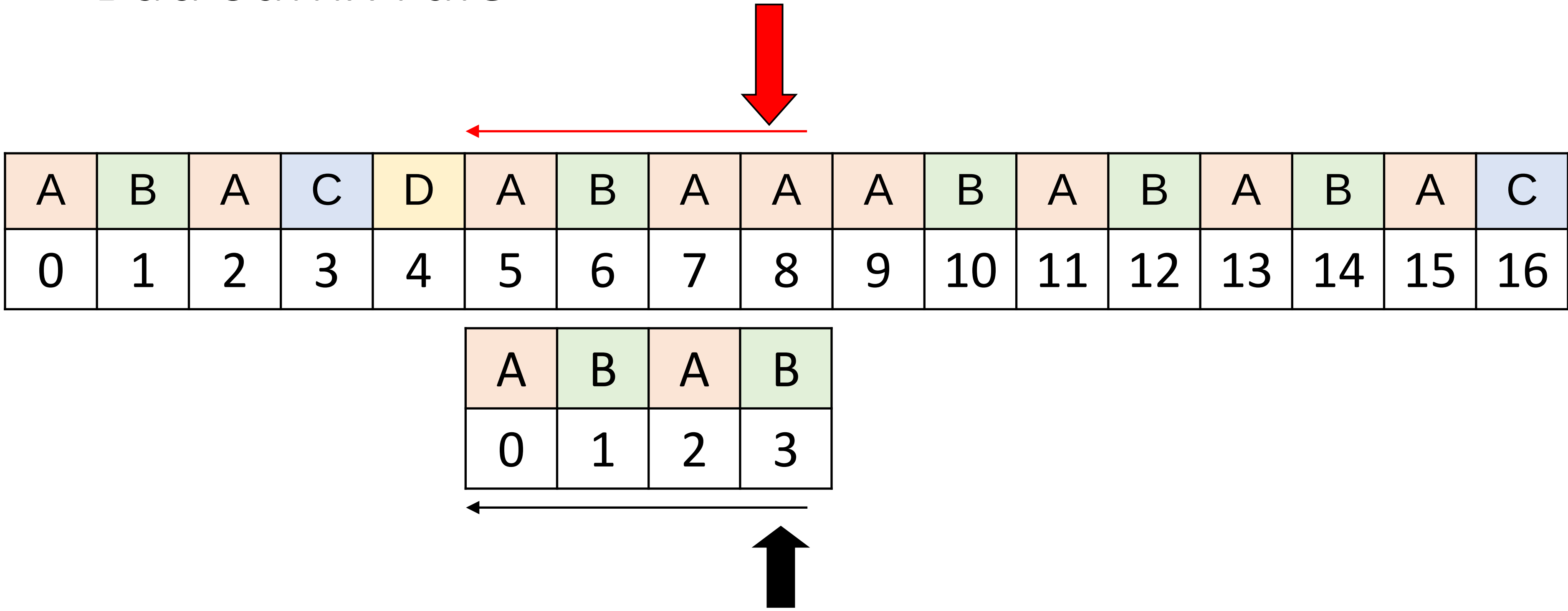
Pattern matching algorithms

- *brute force*
- *Knuth-Morris-Pratt*
- *Boyer-Moore*
- *Rabin-Karp*

Boyer Moore algorithm

- ✓ Bad character rule
- Bad suffix rule
- Good suffix rule (outside the scope of this course)

Bad suffix rule

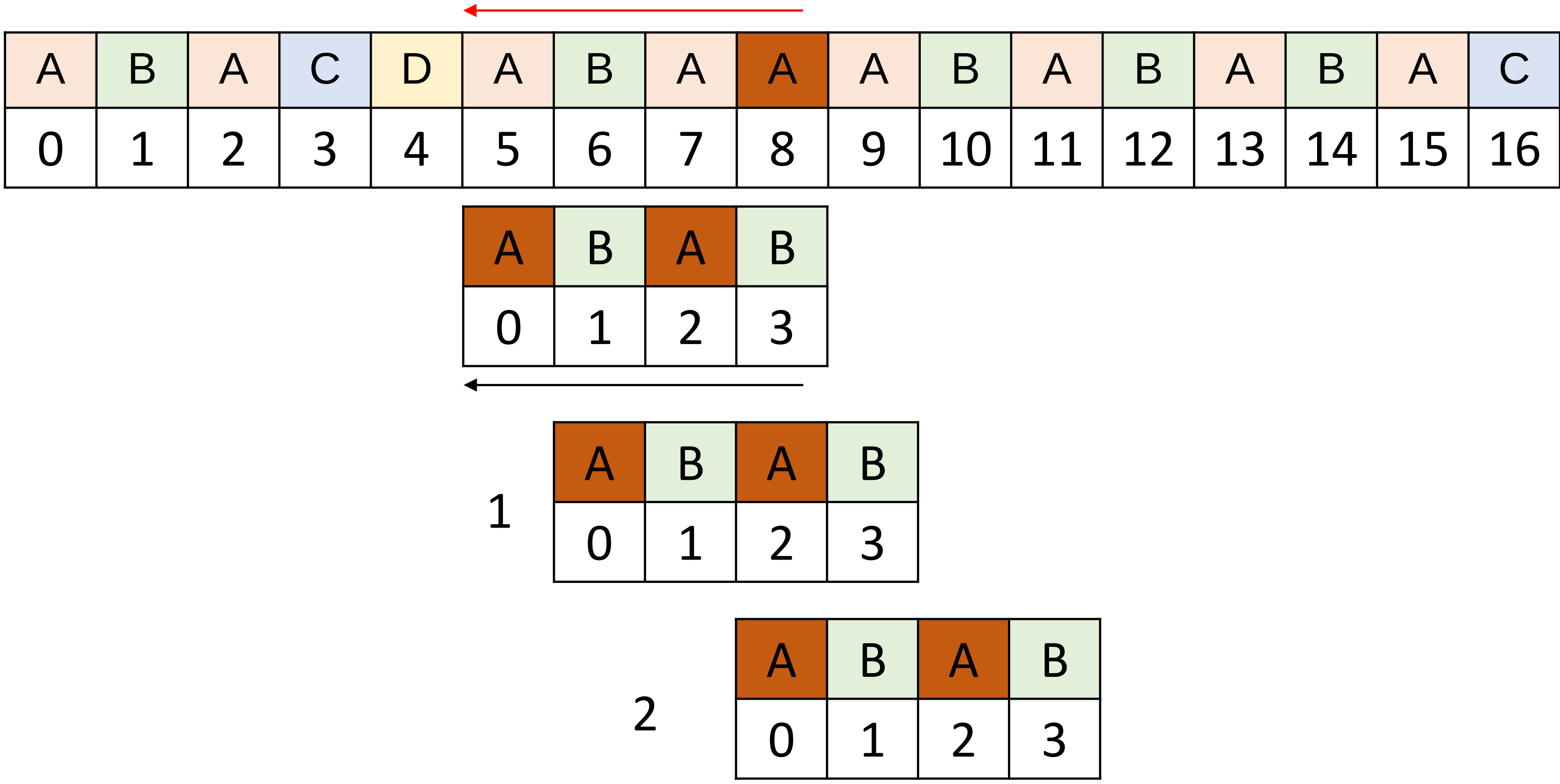


A	B	A	C	D	A	B	A	A	A	B	A	B	A	B	A	C
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

A	B	A	B
0	1	2	3

Now the mismatched character *A* DOES appear in the pattern
Still no need to compare the rest of characters

Bad suffix rule



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

A	B	A	B
0	1	2	3

1

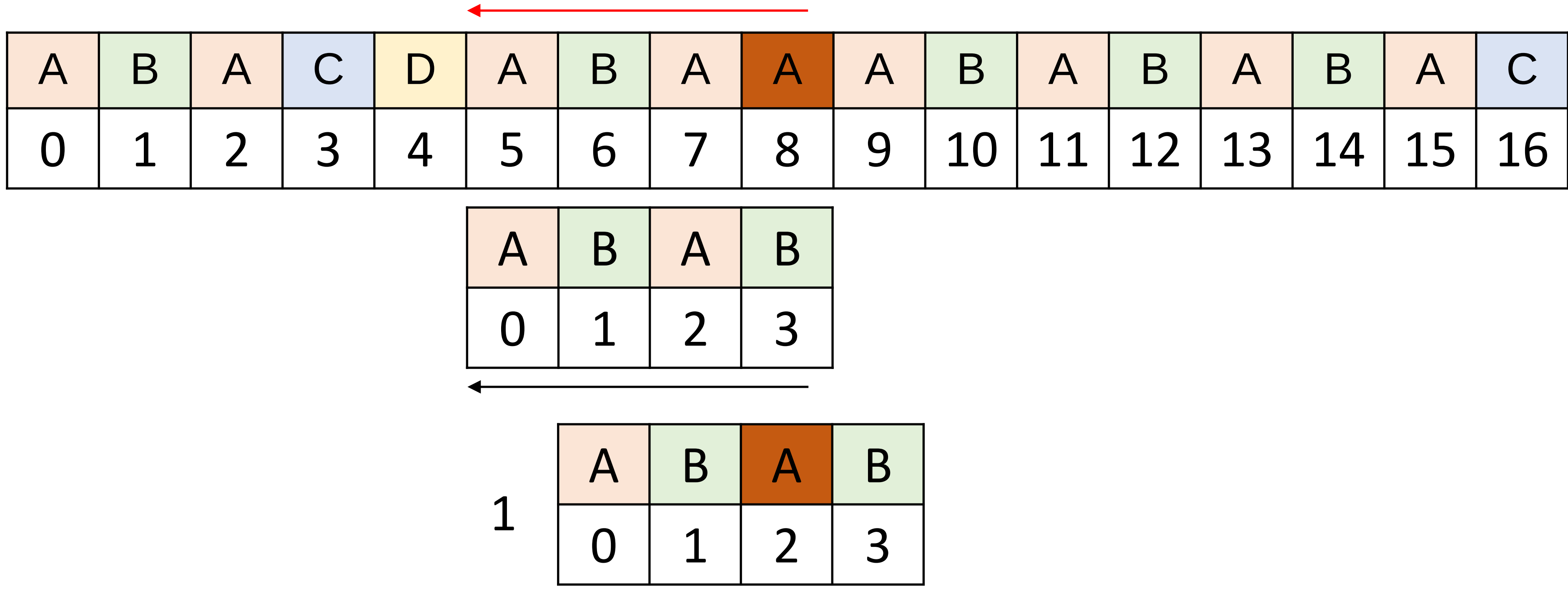
A	B	A	B
0	1	2	3

2

A	B	A	B
0	1	2	3

How far should we slide the pattern for the next comparison?
1 or 2?

Bad suffix rule



- When shifting the pattern to the right, we must make sure not to go farther than where the mismatched character A is first seen (from the right) in P
- For each letter of the alphabet we need to know its last occurrence in P

Pattern preprocessing for the BM algorithm

- How can we figure out how far to skip?
- Preprocess the pattern to create a *right* hash map

```
// position of rightmost occurrence of c in the pattern
right = new HashMap<Character, Integer>();
for (int i=P.length()-1; i>=0; i--) {
   if (right.get(P.charAt(i))==null)
       right.put(P.charAt(i), i);
}
```

- Idea is that if the character does not occur in the pattern we can skip the entire pattern (we return -1 if the entry in the map is *null*)
- The larger the value for P[*j*] in the *right* hash map (the closer this character is to the end of P), the less we can skip

Boyer Moore: code

```
int skip;
for (int i = 0; i <= N - M; i += skip) {
      skip = 0;
      for (int j = M-1; j >= 0; j--) {
          if (P.charAt(j) != T.charAt(i+j)) {
              if (right.get(T.charAt(i))==null)
                   skip = j - (-1);
              else
                   skip = Math.max(1, j - right.get(T.charAt(i)));
                   break;
              break;
          }
      }

       if (skip == 0) return i;     // found
 }
 return -1;                        // not found
```

Note that when j = M-1, and X is not in the pattern then the skip value is ((M-1) - (-1)) = M → this is the bad character rule

Worst-case input

T = AAAAAAAAAAAAAAAAAAAAAAAAAAA

$\xleftarrow{\hspace{3em}}$

P = BAAAAA

- P must be completely compared (*M* char comps, right to left) before we mismatch and skip
- So how far do we skip?

```
skip = Math.max(1, j - right.get(T.charAt(i)));
```

In this case:

- j = 0 (we have moved all the way to the left)
- right['A'] = 5
- This would give a skip of 0-5 (!!!)

- So the actual skip would be 1 – as in the naïve algorithm (this is why we have 1 as a second option)

Running time

T = AAAAAAAAAAAAAAAAAAAAAAAAAAA

$\longleftarrow$

P = BAAAAA

- We will then do the same number of comparisons again – aligning pattern to stat at every position in T
- Thus we do *M* comps, move down 1, *M* comps, move down 1, etc.
- This gives a total of (N-M+1)(M) comparisons or O(MN)

- This is why the actual BM algorithm has another heuristic:
    - Good character rule
- The second heuristic guarantees that the run-time will never be more than O(N)

Pattern matching algorithms

- *brute force*
- *Knuth-Morris-Pratt*
- *Boyer-Moore*
- *Rabin-Karp*

New idea: hashing

- Consider the polynomial hashing scheme we discussed for strings:

hash(s) = $s[0]*A^{n-1} + s[1]*A^{n-2} + \ldots + s[n-2]*A^{1} + s[n-1]*A^{0}$

where A is some prime (31 in JDK)

hash("CAT") = $67*31^{2} + 65*32^{1} + 84 = 66551$

- To search for "CAT" we can thus "hash" all 3-char substrings of our text and test the values for equality
- We need to be able to incrementally update a hash value and not recompute it for M characters, because in this case it will be O(NM)

Hashing of a sliding substring

- Idea is that with each mismatch we "remove" the polynomial part of the leftmost character from the hash value computation and we add the next character from the text to the hash value

Explanation using decimal digits: polynomial of base 10

- P: 1213
- h(P) = $1*10^3 + 2 * 10^2 + 1 *10^1 +3*10^0$

- T: 3526142312136
- h(T[0…3]) = $3*10^3 + 5 * 10^2 + 2 *10^1 +6*10^0$

O(M): we do it only once – for the first substring of T

- T: 3526142312136
- h (T[1…4])
- We should start with h(T[0…3]) and:
  - Remove leading digit $3*10^3$
  - Multiply the result by 10 – moving into the next register
  - Add trailing digit $1*10^0$

That is a constant-time computation

Idea of a rolling hash (digits example)

- P: 1 2 1 3
- T: 3 5 2 6 1 4 2 3 1 2 1 3 6

Hashing pattern P

- $h(P) = 1*10^3 + 2*10^2 + 1*10^1 + 3*10^0 = 1213$

O(M)

Hashing first substring of T

- **3 5 2 6** 1 4 2 3 1 2 1 3 6
- $h(3526) = 3*10^3 + 5*10^2 + 2*10^1 + 6*10^0 = 3526 \quad \neq 1213$

O(M)

- 3 **5 2 6 1** 4 2 3 1 2 1 3 6
- $h(5261) = (h(3526) - 3*10^3)*10 + 1*10^0 = 5260 + 1 = 5261 \neq 1213$

O(1)

- 3 5 **2 6 1 4** 2 3 1 2 1 3 6
- $h(2614) = (h(5261) - 5*10^3)*10 + 4*100 = 2610 + 4 = 2614 \neq 1213$

O(1)

The same thing with each character: but the base of polynomial is some prime

```
long txtHash = hash(T, M);  // hash first M characters of text

for (int i = M; i < N; i++) {
    txtHash -= firstCoef*T.charAt(i-M);  //Remove leading digit
    txtHash = txtHash*polBase;           //move register
    txtHash += T.charAt(i);              //add trailing digit

    // check if that actually was a match
    int startInd = i - M + 1;
    if ((patHash == txtHash) && check(T, startInd))
        return startInd;
}

// no match
return -1;
```

Rabin Karp: notes

- We are moving from left to right in the text
- We do not compare characters: we compare hash values of each substring of the text T of length *M* with the hash of a pattern P
- We only compute the full hash once: for T[0…M-1]
- After that we update the hash value of the next substring in time O(1)

- In the robust implementations of the algorithm we use modulo of hashing results to avoid integer overflow
- That is why the simplified version presented in demo can only work for fairly short patterns

Where there is a hash – there are collisions

- So what happens if two different substrings hash to the same hash code?
- The hash values would match but in fact the strings would not! They could be not even similar!

- To make sure that we found the pattern, if hash values match – we could verify the result with a char-by-char test

Two versions of the Rabin Karp algorithm

- If we don’t check for collisions:
    - Algorithm is guaranteed to run in O(N) time
    - Algorithm is highly likely to be correct
    - But it could also produce false positive if a collision occurs

Text calls this a *Monte Carlo* version

Recap:
Monte Carlo algorithms: produce results that are not always fully correct (close to being correct with some probability), but the runtime is guaranteed

Two versions of the Rabin Karp algorithm

- If we do check for collisions:
  - Algorithm is guaranteed to be correct
  - Algorithm is highly likely to run in O(N) time
  - What would worst case be and why? O(MN)
- Text calls this the *Las Vegas* version

Recap:
Las Vegas algorithms always produce the correct result but they guarantee the runtime only with some probability

Pattern matching algorithms: runtime

- Brute force (naïve algorithm):
  - O(MN) – but rarely reaches this in practice
- KMP:
  - O(N) – always
- Boyer-Moore:
  - Can be sublinear < N
  - With only a bad suffix rule: O(MN)
  - With the good suffix rule: guarantees O(N)
- Rabin-Karp:
  - O(N) expected
  - O(MN) worst case (Las Vegas version, very unlikely)