Data Structures and Algorithms

Lesson 12. String Algorithms and Techniques

Alexandra Ciobotaru

12th of May 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- **Lesson 8. Graphs.**
- **Lesson 9. Searching.**
- **Lesson 10. Sorting.** Mid-term evaluation quiz (30%)
- **Lesson 11. Selection Algorithms.**
- **Lesson 12. String Algorithms and Techniques.**
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

What you'll learn

- Brute-force algorithm for pattern matching (naïve method)
- Rabin Karp algorithm (difficulty level: medium)
- Knuth Morris Pratt algorithm (difficulty level: hard)

String Notations

- **Strings** - a sequence of objects, mainly a sequence of characters.
  - we need to store the data and operations that have to be applied to them
  - `string = "this is a data structures class"`
- **Substring** - a sequence of characters that's part of the given string.
  - `substring = "data structures"`
- **Subsequence** - is a sequence of characters that can be obtained from the given string by removing some of the characters from the string but by keeping the order of occurrence of the characters.
  - `subsequence = "data strctrs"`
- **Prefix** – a substring of the string that it is present at the starting of the string.
  - `prefix = "this is"`
- **Suffix** – a substring that is present at the end of the string.
  - `suffix = " class"`

Pattern Matching Algorithms

- A pattern matching algorithm is used to determine the index positions where a given **pattern** string (P) is matched in a **text** string (T).
- It returns "pattern not found" if the pattern does not match in the text string.
- For example, for the given:
  - string `(s) = "packt publisher"`
  - and the pattern `(p) = "publisher"`

  The pattern matching algorithm returns the index position where the pattern is matched in the text string.

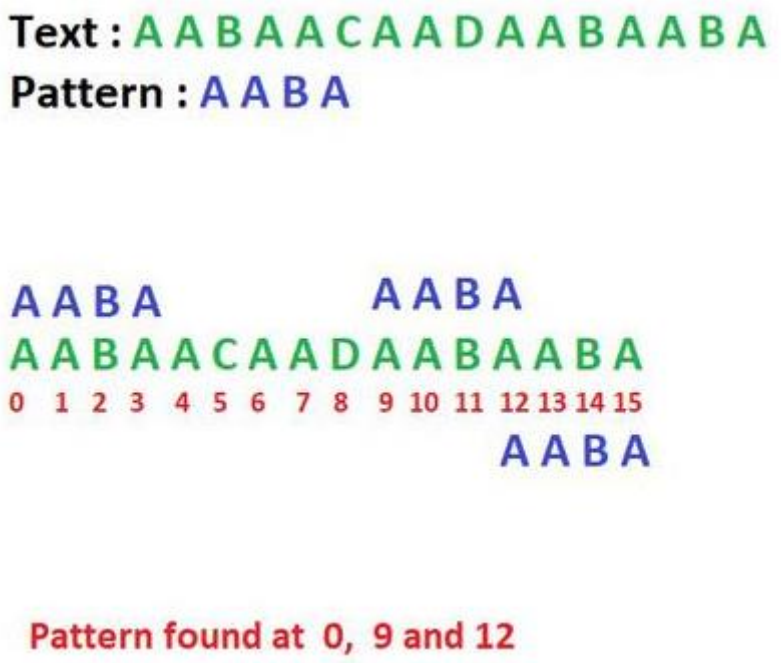
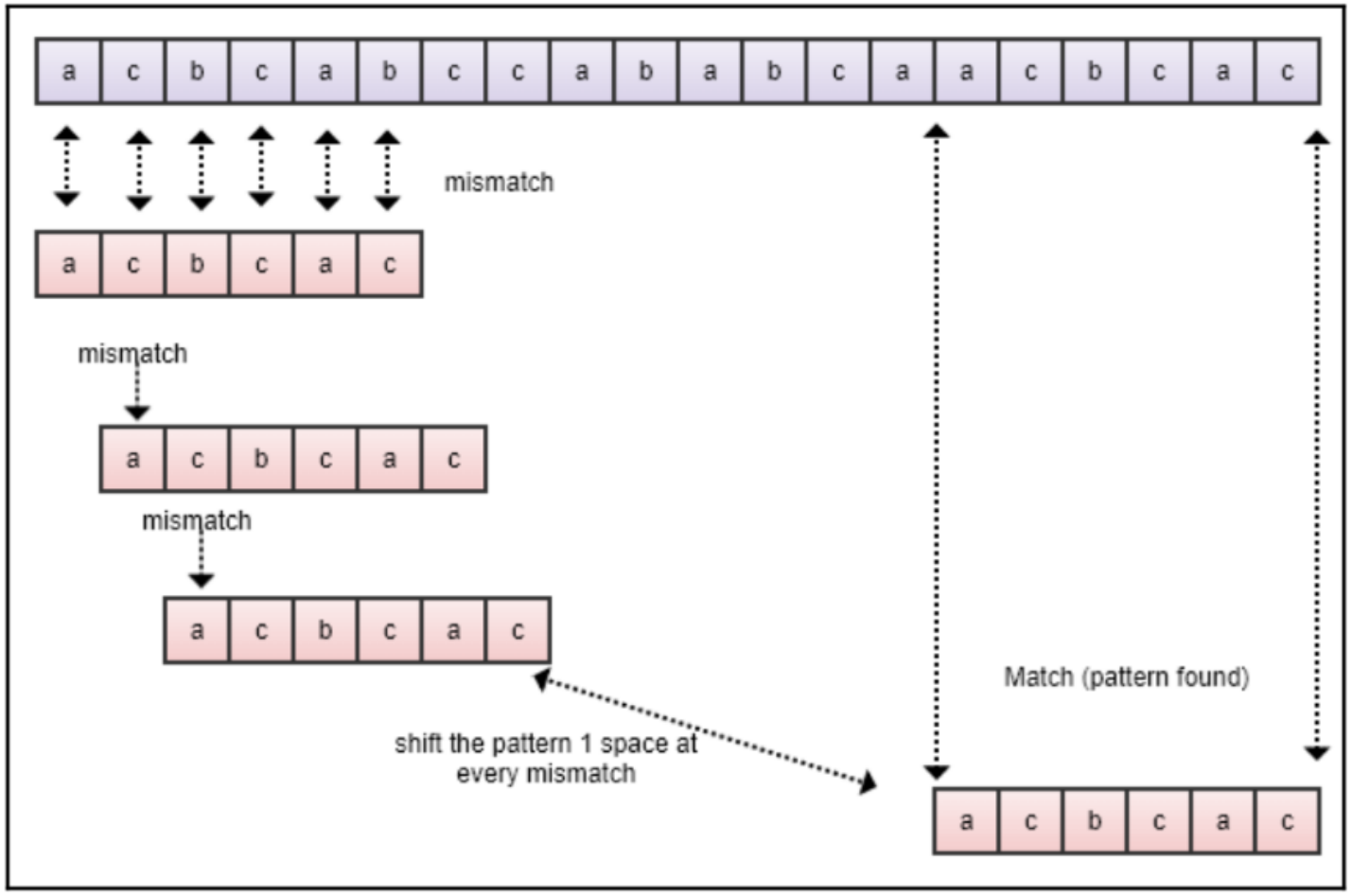


Image source: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

Brute-force Algorithm

- We test all the possible combinations of the input pattern in the given string to find the position of the occurrence of the pattern.
- This algorithm is very *naive* and is not suitable if the text is very long.
- We start by comparing the characters of the pattern and the text string one by one, and if all the characters of the pattern are matched within the text, we return the index position of the text where the first character of the pattern is placed.
- If any character of the pattern is mismatched with the text string, we shift the pattern by one place.
- We continue comparing the pattern and text string by shifting the pattern by one index position.

Brute-force Algorithm



Even if the mismatch is at the end of the pattern, we still have to shift with
one position –> this is a waste of time (too much useless extra work).

Worst Case: O(mn)

String (n)	a	a	a	a	a	a	a	b
Pattern (m)	a	a	a	b				

Implementing Brute-force

```
def brute_force(text, pattern):
    l1 = len(text)      # The text which is to be checked for the existence of the pattern
    l2 = len(pattern)   # The pattern to be determined in the text
    i=0                 # looping variables are set to 0 (I to iterate through text, j to iterate through pattern)
    j=0
    flag = False        # If the pattern doesn't appear at all, then set this to false and execute the last if statement
    while i < l1:       # we iterate from the 0th index of text
        j = 0           # we reinitialize j with 0
        count = 0       # Count stores the length up to which the pattern and the text have matched
        while j < l2:
            if i+j<l1 and text[i+j] == pattern[j]:  # statement to check if a match has occured or not
                count += 1                          # if the statement evaluates to true, then update count
            j += 1                                  # we shift the pattern to the right
        if count == l2:                             # if total number of successful matches is equal to the pattern length
            print("\nPattern occours at index",i)   # we print the starting index of the successful match
            flag = True                             # if the matching occurs once, we set this flag to True
        i += 1                                      # we increment i so we can continue the search in text
    if not flag:                                    # If the pattern doesn't occurs even once, this statement gets printed
        print('\nPattern is not at all present in the array.')

brute_force('acbcabccababcaacbcaabacbbc','acbcaa')
```

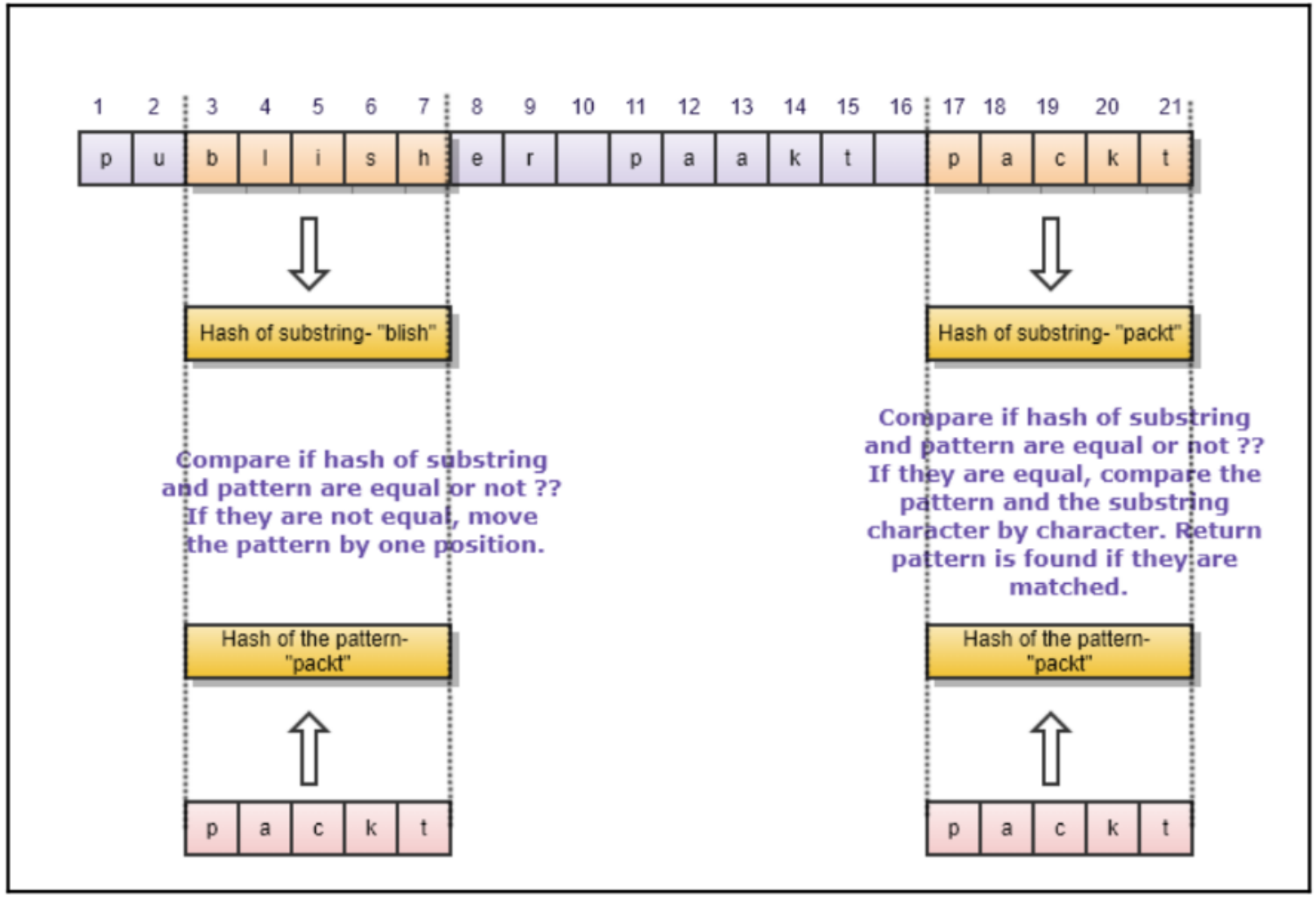
Rabin-Karp Algorithm

- The Rabin-Karp pattern matching algorithm is an improved version of the brute-force approach.
- The performance of the Rabin-Karp algorithm is improved by reducing the number of comparisons with the help of hashing
  - the hashing function returns a unique numeric value for a given string.
- Instead of unnecessary comparisons, character by character, the hash value of the pattern is compared with the hash of the substring of the text string (of the same length as the pattern) all at once.
- If the hash values are not matched, the pattern is moved one position (there is no more the need to compare all the characters of the pattern one by one).
- This algorithm is based on the concept that *if the hash values of the two strings are equal, then it is assumed that both of these strings are also equal*.
- But the main problem with this algorithm is that ***there can be two different strings whose hash values are equal*** – this is called SPURIOUS HIT (a case in which the algorithm might not work) and is due to collision in hashing.
  - To avoid this problem, after matching the hash values of the pattern and the substring, we ensure that the pattern is actually matched by comparing them character by character.

Rabin-Karp Algorithm

- Let:
  - `n` = length of the text where we search
  - `m` = length of the pattern
- Steps:
  1. We compute the hash value of the pattern (of length `m`) and the hash values of all possible substrings (of length `m`) -> total no. of possible substrings is `n−m+1`;
  2. We compare the hash value of the pattern with **all** the hash values of the substrings, one by one;
  3. If the hash values are not matched, we move the pattern by one position;
  4. If the hash value of the pattern and the hash value of the substring matches
     ✓ we compare the pattern and the substring character by character;
  5. We continue steps 2-4 until we reach the end of the text string.

Rabin-Karp Algorithm



T = “publisher paakt packt”
P = “packt”

First we compute the hash of P and the hashes of all substrings in T.

We compare hash of P with hash of “publi” –> no match –> we move P by one location.

We always move to the right when patterns do not match.

We find a match at index 17 –> now we compare P and the substring element by element.

The Hashing Function

- We will use the sum of all the ordinal values of the characters in the string as the hash function.

```
def generate_hash(text, pattern):
    ord_text = [ord(i) for i in text]          # stores unicode value of each character in text
    ord_pattern = [ord(j) for j in pattern]    # stores unicode value of each character in pattern
    len_text = len(text)                       # stores length of the text
    len_pattern = len(pattern)                 # stores length of the pattern
    # we store the total no. of possible substrings of the length of the pattern
    len_hash_array = len_text - len_pattern + 1
    hash_text = [0]*(len_hash_array)           # will store the hash values for all possible
substrings. For now we initialize it with 0.
    hash_pattern = sum(ord_pattern)            # we compute the hash of the pattern
    # we start a loop that will run for all the possible substrings in text, for which we compute
hashes
    for i in range(0,len_hash_array):          # step size of the loop will be the size of the pattern
        if i == 0:                             # Base condition
           # initial value of the hash function
           hash_text[i] = sum(ord_text[:len_pattern])
        else:
           # computing next hash value using previous value
          hash_text[i] = ((hash_text[i-1] - ord_text[i-1]) + ord_text[i+len_pattern-1])
     return [hash_text, hash_pattern]
```

Implementing Rabin-Karp

```
def Rabin_Karp_Matcher(text, pattern):
    text = str(text)                                          # convert text into string format (for no.)
    pattern = str(pattern)                                    # convert pattern into string format (for no.)
    hash_text, hash_pattern = generate_hash(text, pattern)    # we generate hash values using the previous function
    len_text = len(text)                                      # length of text
    len_pattern = len(pattern)                                # length of pattern
    flag = False                                              # checks if pattern is present at least once or not at all
    # we starts a loop that will run for the length of hash_text (for all possible substrings)
    for i in range(len(hash_text)):
        if hash_text[i] == hash_pattern:                      # if the hash values match
            count = 0                                         # count stores the total characters up to which both are similar
            # if they match, we compare the pattern and the substring character by character
            for j in range(len_pattern):
                if pattern[j] == text[i+j]:                   # checking equality for each character
                    count += 1                                # if values are equal, we update the count value,
to keep track of how many characters match in the pattern and the substring
                else:
                    break
            if count == len_pattern:    # if count is equal to length of pattern, it means match has been found
                    flag = True                               # update flag accordingly
                    print('Pattern occours at index',i)
    if not flag:                    # if pattern doesn't match even once, then this if statement is executed
        print('Pattern is not at all present in the text')
```

Knuth-Morris-Pratt Algorithm

- The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we may already know some of the characters in the text of the next window.
- We can take advantage of this information to avoid matching the characters that we know that will match anyway.

```
txt = "AAAAABAAABA"
pat = "AAAA"
```

- We compare first window of txt with pat

```
txt = "AAAAABAAABA"
pat = "AAAA"  [Initial position]
```

- We find a match. This is same as Naive String Matching. In the next step, we compare next window of txt with pat.

```
txt = "AAAAABAAABA"
pat =  "AAAA" [Pattern shifted one position]
```

- This is where KMP does an optimization: in this second window, we only compare the fourth A of pattern with the fourth character of the current window of text, in order to decide whether the current window matches or not. Since we know the first three characters will anyway match, we skip matching the first three characters.

The prefix function

- But how do we know how many characters we need to skip??
- We store the matchings in the *prefix table,* made with a *prefix function* which does a preprocessing to the pattern, by finding the pattern in the pattern itself

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2	0	0	1	2

- The main idea behind the KMP algorithm is to detect how much the pattern should be shifted, based on the overlaps in the patterns.

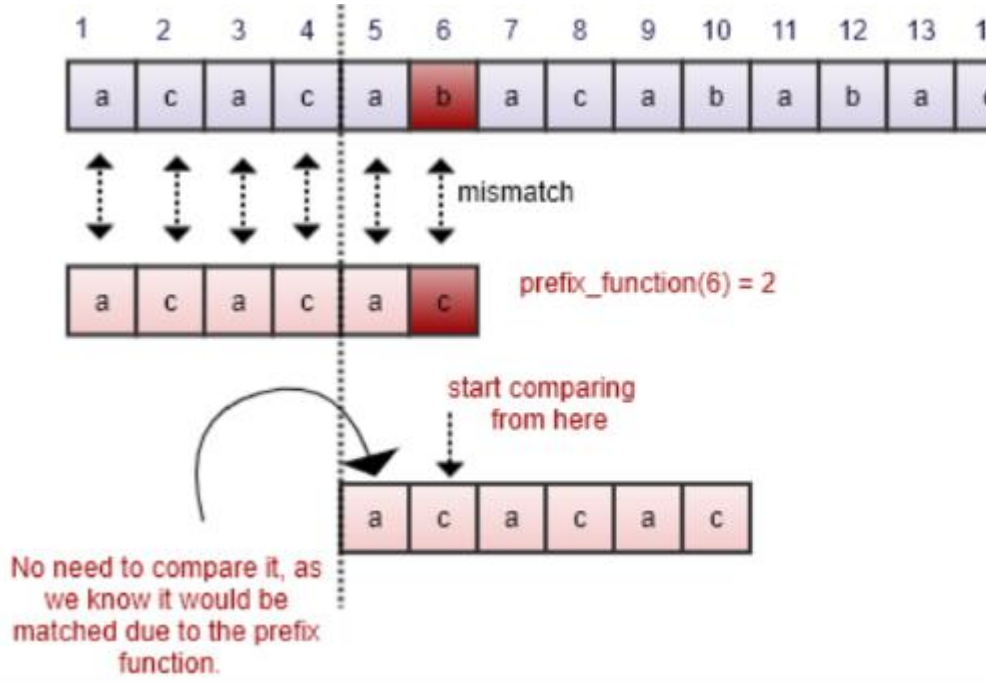
Understanding KMP

- The algorithm works as follows:
  1. We precompute the prefix table for the pattern and initialize a counter, q, that represents the number of characters that matched
  2. We start by comparing the first character of the pattern with the first character of the string, and if this matches, then we increment the counter, q, for the pattern and the counter for the text string, j, and we compare the next character.
  3. **If there is a mismatch, then we assign the value of the precomputed prefix function for q to the index value of q**.
  4. We continue searching the pattern in the text string until we reach the end of the text, that is, if we do not find any matches. If all of the characters in the pattern are matched in the text string, we return the position where the pattern is matched in the text and continue to search for another match.
- So the KMP algorithm preprocesses the pattern so that the pattern can be shifted by more than one (like the naïve approach).

Understanding KMP

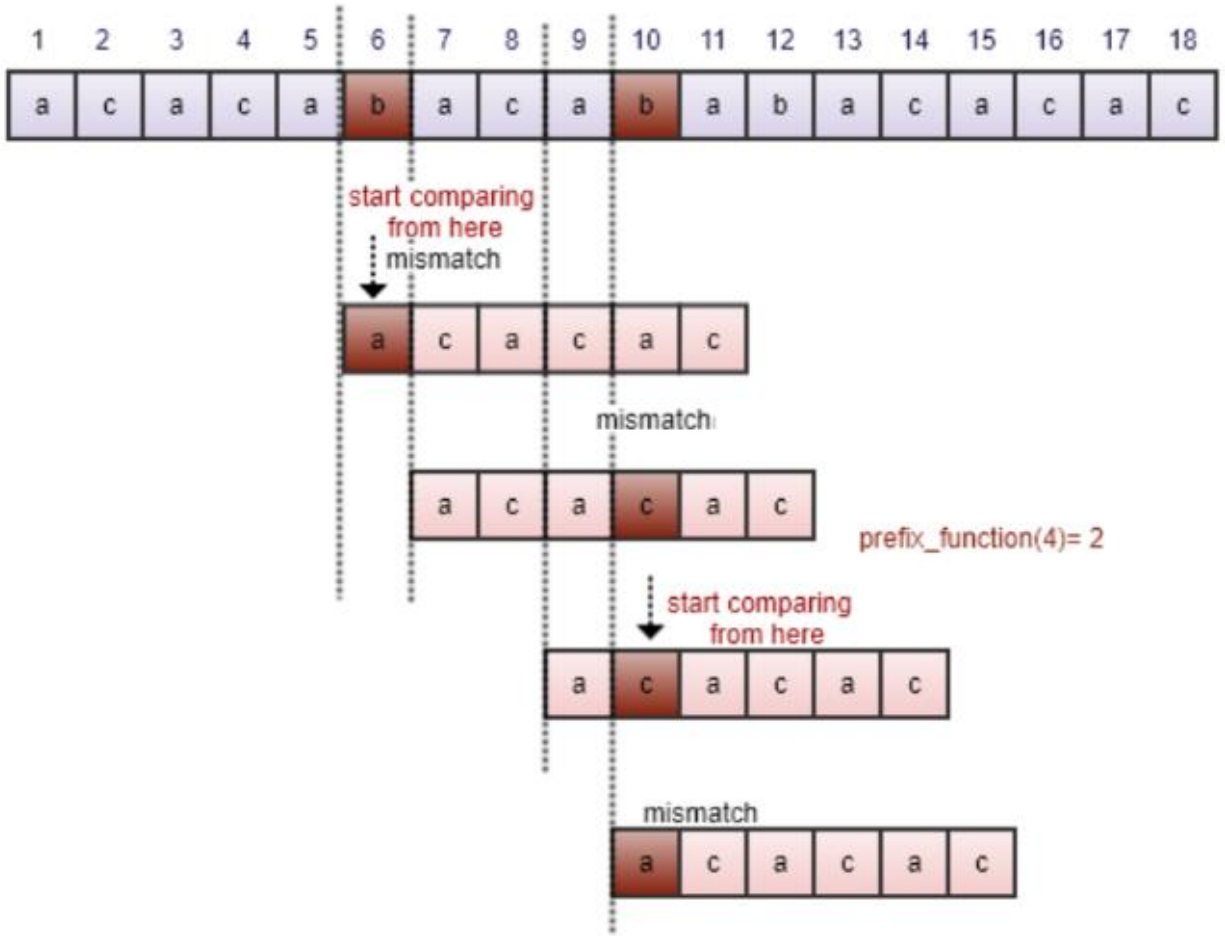
- Suppose we have this pattern
- And we want to match it

Index	1	2	3	4	5	6
Pattern	a	c	a	c	a	c
Prefix_function	0	0	1	2	1	2



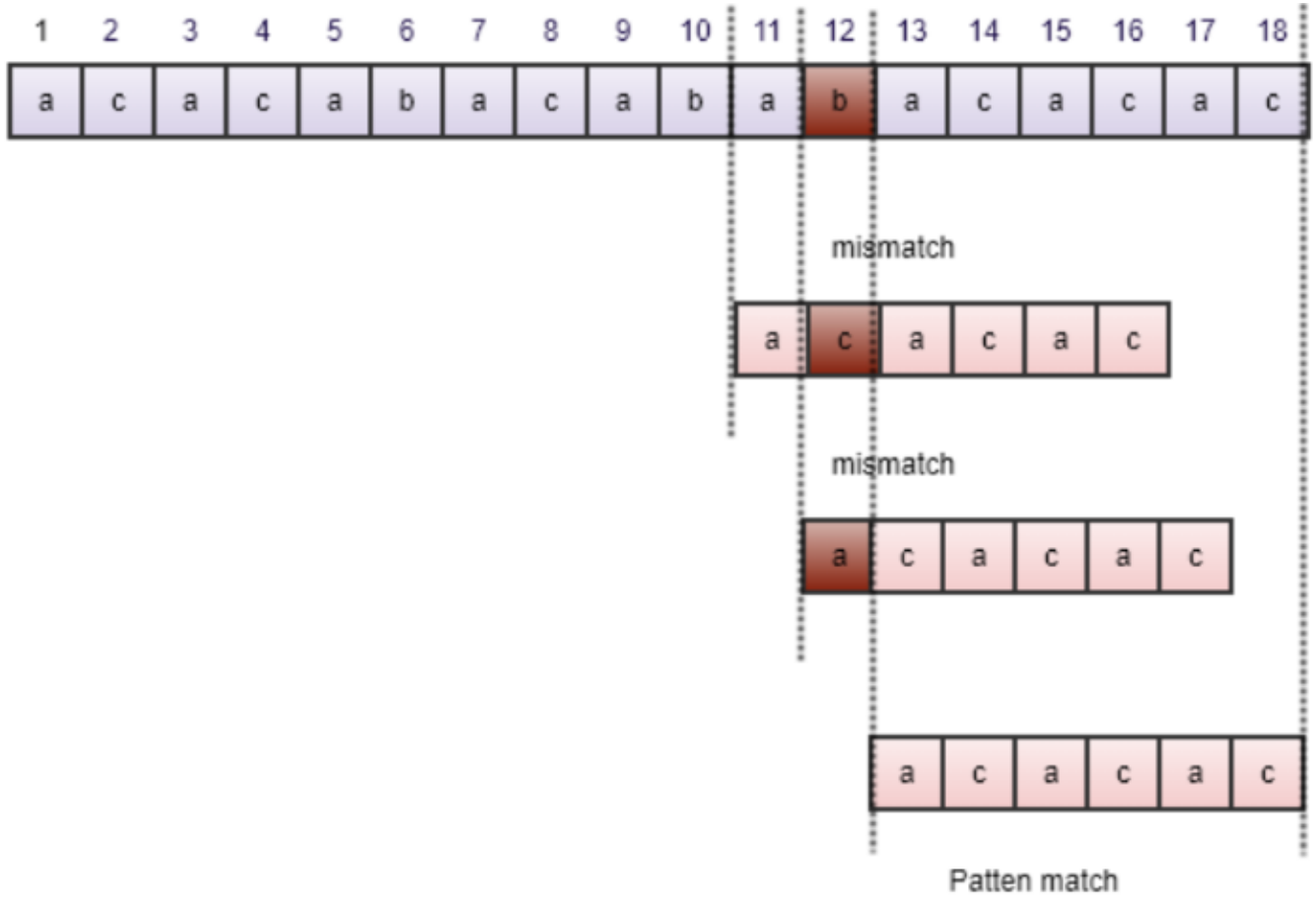
- We start by comparing the first character of the pattern with the first character of the text, and continue until we find a mismatch or we have compared the whole pattern.
- We find a mismatch at index position 6, so now we have to shift the pattern -> we find the no. of shifts to take by using the prefix_function (prefix_function (c) = 2)
- So, we start comparing the pattern from index position 2 of the pattern (we compare *c* and *b*)
- *c* with *b* do not match, so we shift the pattern 1 position to the right (prefix_function(c) = 0)

Understanding KMP



- So *c* with *b* do not match, so we shift the pattern 1 position to the right (prefix_function(c) = 0)
- *b* and *a* do not match, so we shift the pattern 1 position to the right (prefix_function(a) = 0)
- Next all characters match… up to index 10 in text, where we find the first mismatch, at *c*
- prefix_function(c) = 2 so we shift the pattern at its index position of 2
- *b* with *c* do not match, so we shift the pattern 1 position to the right (prefix_function(c) = 0)
- *b* with *a* do not match, so we shift the pattern 1 position to the right (prefix_function(a) = 0)

Understanding KMP



- a and a match, b and c give a mismatch
- prefix_function(c) = 0 so we shift the pattern up to index 0 in pattern
- b and a do not match, so we shift the pattern 1 position to the right (prefix_function(a) = 0)
- From this point, everything matches.

Worst Case: O(m + n)
m = len(text)
n = len(pattern)

Implementing the prefix function

```
def pfun(pattern):
    n = len(pattern)          # length of the pattern
    prefix_fun = [0]*(n)      # here we store the computed values of the prefix function. We initialize all elements of the list to 0
    k = 0  # the prefix function for the first element of the pattern
    for q in range(2,n):
        # a nested loop that is executed until we processed the whole pattern
        while k>0 and pattern[k+1] != pattern[q]:
            k = prefix_fun[k]
        # If the k^th element of the pattern is equal to the q^th element
        if pattern[k+1] == pattern[q]:
            k += 1                              # we increment k by 1
        # the value of k is the computed value of the prefix function and we put it at its correspondent position q
        prefix_fun[q] = k
    return prefix_fun

print(pfun('abcsedfabf'))
```

Implementing KPM algorithm

```
def KMP_Matcher(text,pattern):
    m = len(text)
    n = len(pattern)
    flag = False
    text = '-' + text                          # append dummy character to make it 1-based indexing
    pattern = '-' + pattern                    # append dummy character to the pattern also
    prefix_fun = pfun(pattern)                 # generate prefix function for the pattern
    q = 0
    for i in range(1,m+1):
        # while pattern and text are not equal, decrement the value of q if it is > 0
        while q>0 and pattern[q+1] != text[i]:
            q = prefix_fun[q]
        # if pattern and text are equal, update value of q
        if pattern[q+1] == text[i]:
            q += 1
        if q == n: # if q is equal to the length of the pattern, it means that the pattern has been found.
            print("Pattern occurs with shift",i-n)     # print the index, where first match occurs.
            flag = True
            q = prefix_fun[q]
    if not flag:
        print('\nNo match found')

print(KMP_Matcher("acesta este un test, un test important", "test"))
```

Thank you!