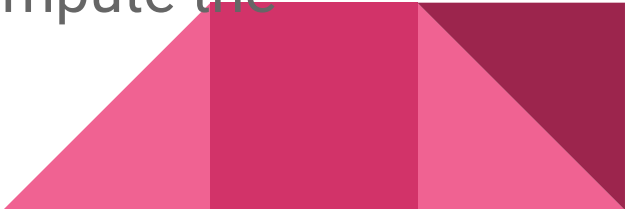


Pattern Matching Algorithms

Motivation :

- We need various string algorithms to **process the text** which consume resources as optimally as possible.
- We studied various **compression algorithms**, which can be used to drastically reduce the space complexity.
- For implementing Ctrl+F (FIND) functionality, we need an algorithm which processes the entire text as quickly as possible to find all the instances of the pattern in the text. Algorithm shouldn't perform repeated scans of the same piece of text in order to improve time complexity of the search or, use intelligent filters to check before actually confirming the presence of the pattern in the text.
- Recommendation system which many search engines have, can also be simulated by using dictionaries which is used to store a set of strings. Though the ones being actually used in real time are incredibly massive, and involve ML improvisation, but this can be implemented in the scenarios having less number of permissible words like auto complete functionality in many code editors

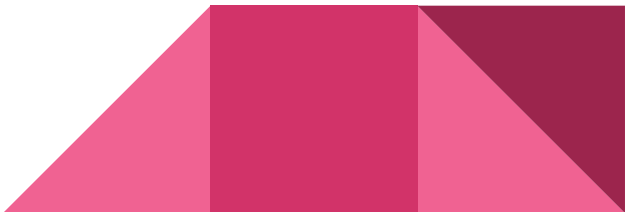
- Sometimes, we may have to implement the functionality of spelling correction and “do you mean?” (like in Google search engine)
 - For this, we need an efficient algorithm which digs into the space of possible strings and extract the ones which are close enough to the string entered in the tab.
 - For this algorithm to be efficient enough, we have to give optimal definition which suffices the requirement, to the proximity between the given 2 strings and also an algorithm which consumes optimal resources to compute the proximity as per the definition.
- 

Index

- Naive
- Karp-Rabin
- Bitap-Algorithm
- Dictionary Implementation
- KMP
- Automaton Algorithm
- Manacher's Algorithm
- Z-Algorithm



Brief Overview

- As the name suggests, this algorithm is Brute Force algorithm for detecting pattern in the text. If mismatch is found, the algorithm simply goes to the next window without using the information obtained till the characters matched before the instance of mismatch.
 - Time Complexity is $O(m*n)$ where m is the size of the pattern and n is the size of the text
 - Space complexity of the algorithm is $O(1)$ as it doesn't use any array or other data structure whose worst case space occupancy varies as a function of input size.
- 


Karp Rabin Algorithm

Brief Overview

- In naive algorithm, we simply check each and every window without applying any constant time filter check.
- But in this algorithm, we apply this filter check to see whether the window is equal to the pattern or not. For this the concept of rolling hash is used.
- In rolling hash, hashing scheme is very important because if the hash function is poor i.e., it has minimum bijection or it is very complicated i.e., they need more time and space resources then the scheme doesn't serve the very purpose of hashing



Cookbook for this algorithm

- Step 1: Compute the hash value of the pattern
 - Step 2: Using the same scheme, compute the hash value of the first window of size of the pattern. Compare the hashes of this window and that of pattern. If the values are equal, then we have to compare this window and pattern character by character.
 - Step3: From previous window's hash value, we have to compute the hash value of the current window in an efficient manner and like step 2, we have to compare this with pattern's hash and do the same process of checking character by character if necessary
- 

Role of hash function's efficiency

- As we see if collisions increase then many windows may have the same hash value as that of pattern and we may have to perform many character by character comparisons thus, not improving time complexity.
- And here every window is virtually like a roll. While moving from one window to the next one, we are simply popping out the earliest character and pushing in the latest character so $m-1$ elements will still be present in the roll. So, if we have such a hashing scheme which computes the hash value of a window from its previous contiguous window in constant order time, then the algorithm efficiency will be high.



Intuition behind hash function

- In hexadecimal system, for example number 16A is 362 in decimal system and in hexadecimal system, there is only one way of representing 362 similarly 16A also means only 362 in decimal. So, this shows perfect bijection, which is needed for a good hash function.
- For example consider 2 hexadecimal numbers 16A and 6AD. 16A can be expanded in decimal system as $(1 \cdot 16^2) + (6 \cdot 16) + 10$. Similarly, 6AD is $(6 \cdot 16^2) + (10 \cdot 16) + 13 = 1709$. If we have to calculate 6AD from 16A, we can do $((362 - (1 \cdot 16^2)) \cdot 16) + 13 = 1709$.




Intuition behind hash function

- So formally, rehash function is,
- $hash_curr = (((hash_pre) - ((out_going_digit) * pow(base, window_size - 1))) * (base)) + (in_coming_digit)$
- If we compute $pow(base, window_size - 1)$ and store it, the rest of the function can be computed in constant time.
- So this scheme has both characteristics of a good hashing scheme.



Collisions and practical constraints

- Though this hashing scheme looks the best, while programming this algorithm, practical constraints come into play. Like system not being able to handle large numbers in the order of even 10^{19} .
 - So, for avoiding overflow, we have to apply modulus of some number in order to ensure that all outputs are capped below a certain number. So, we should apply modulus of the largest number possible in order to allow vivid numbers covering all the possible space for checking collisions as much as possible. As number of possible characters are limited, we need not worry about the base. Base of the numbers would be 256 as there are only these many possible characters.
- 

Complexity Analysis

- Best case time time complexity: Linear $O(n+m)$
- Worst case time complexity: Quadratic $O(m*n)$
- Example: Text="AAAAAAAAAAAA" Pattern="AAA". Number of steps taken is $11*3=33$. Because, every consecutive window has got the same string. So, no matter how much efficient the function is, it will run in quadratic time because all same strings yield the same hash function no matter what the hash function is.
- Conclusion: This algorithm is better than naive algorithm, if the pattern is sparsely present in the text as, there will be less contiguous windows matching with the pattern.
- Space complexity: Constant, as no data structure is being used.

KMP Algorithm

The Mother of all definite linear time pattern searching algorithms

Brief Overview

- Unlike Naive algorithm, this algorithm is of linear time complexity and linear space complexity
- This algorithm has 2 phases: pre-processing phase and text processing phase
- This algorithm uses the concept of largest proper prefix and suffix.
- This algorithm basically finds the largest proper prefix , which is also a proper suffix of the prefix of the pattern, of the prefix of the pattern, which matched with the text.



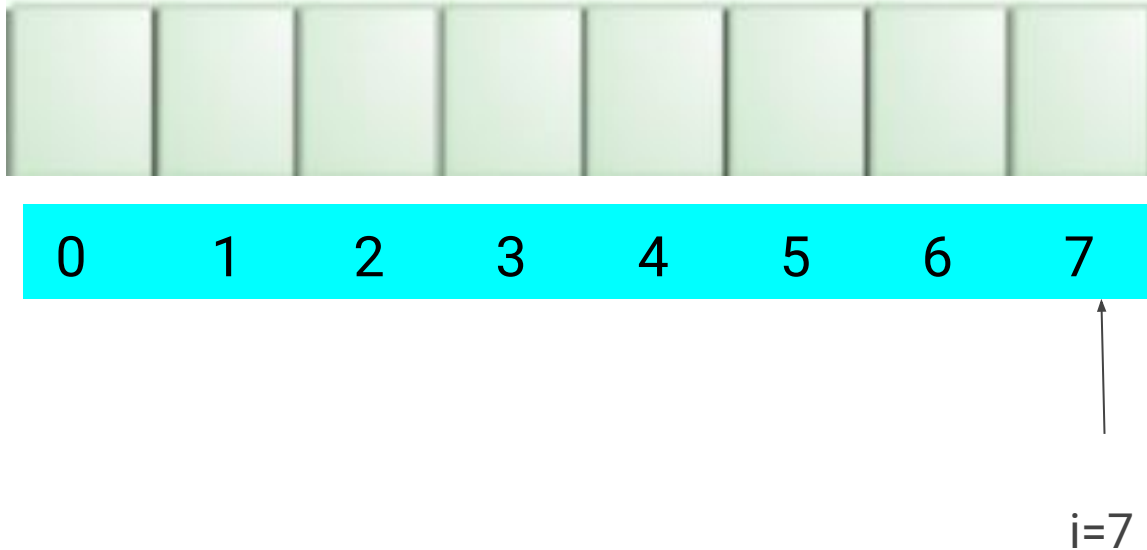
Terms and Definitions

- Proper Prefix of a string: Any prefix of the given string except the entire string itself. For example, ABCDE has A, AB, ABC, ABCD as proper prefixes.
- Proper Suffix of a string: Any suffix of the given string except the entire string itself. For example, ABCDE has E, DE, CDE, BCDE as proper suffixes.
- Lps: Longest proper prefix which is also a proper suffix of the given string, which is represented as the succeeding index of the Lps of the same string, in the algorithm. Lps of the first prefix is taken to be 0. For example, AAABAAA has A, AA, AAA, AAAB, AAABA, AAABAA, as proper prefixes and A, AA, AAA, BAAA, ABAAA, AABAAA as proper suffixes and we can say that AAA is the largest string common between these 2 sets. In this algorithm, Lps i.e., AAA of this string, is represented as 3 as 3 is the index of the letter just after AA'A' (in ' ')

- Pi table: The final table containing lps of each and every prefix of the string. In Pi table, key value pairs are of the format <integer, integer> where key indicates the last index of the prefix and its corresponding value indicates the LPS of the string, which is a prefix with last index as the key. For example, AAABAAA has lps=3. So, it is represented as $Pi[6]=3$ as 6 is the last index of the prefix and 3 is the string's Lps. Similarly, Lps of AAAB is stored in $Pi[3]$.

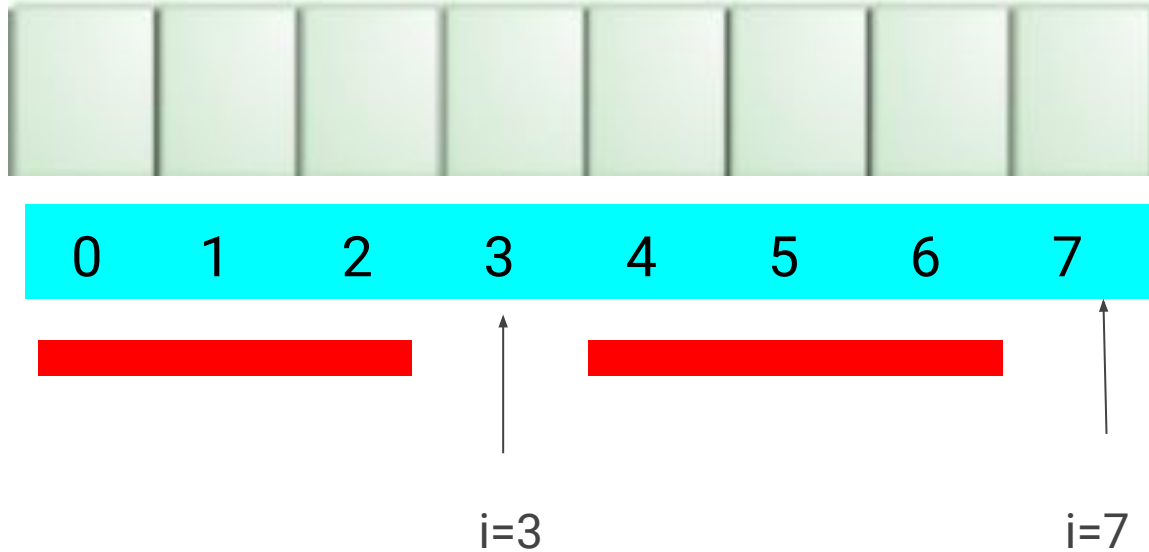


Preprocessing Phase: Construction of Pi Table



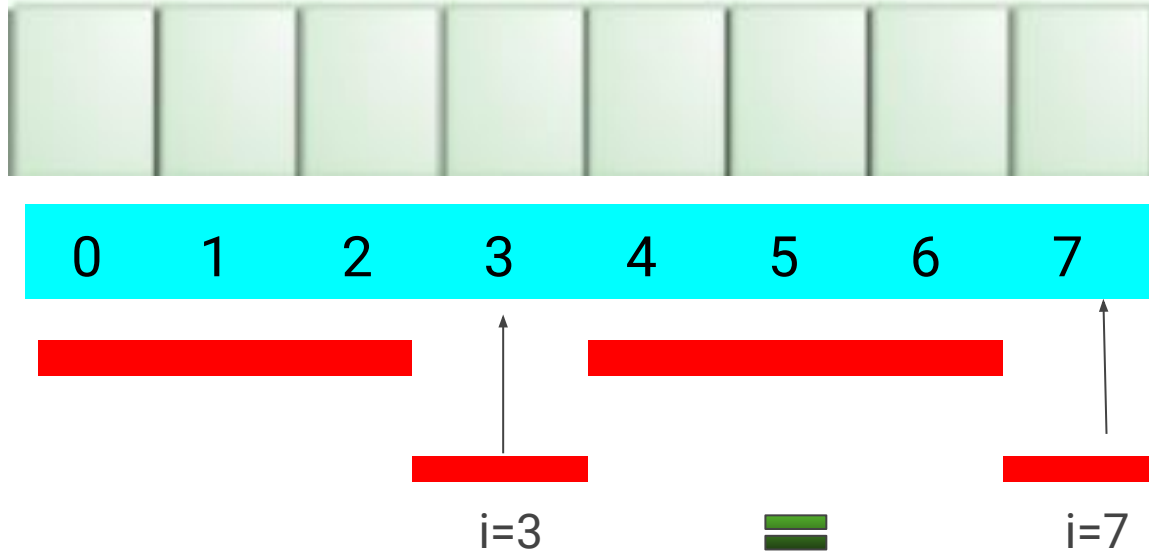
For knowing LPS at $i=7$, we have to consider LPS at $i=6$ and sometimes even before that

Preprocessing Phase: Construction of Pi Table



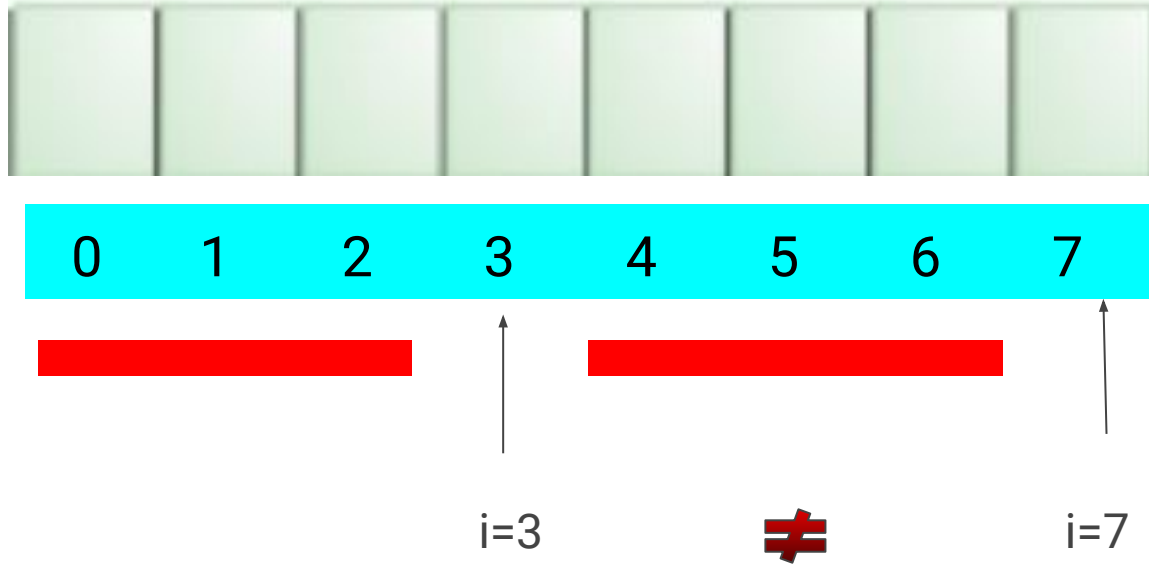
Red bars indicate LPS of prefix ending at $i=6$

Preprocessing Phase: Construction of Pi Table

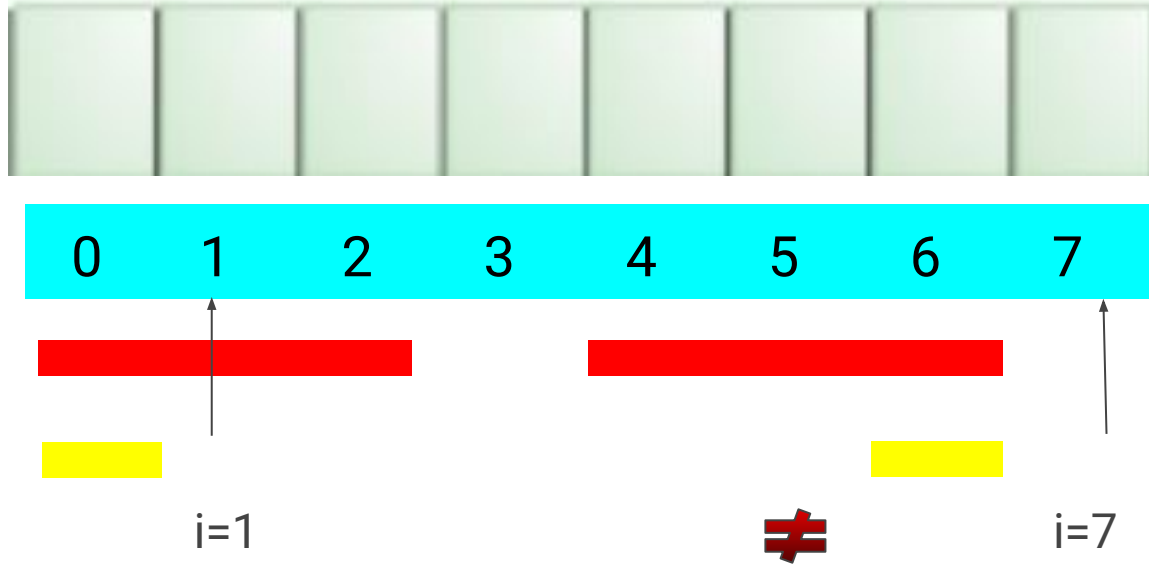


If equal, then 0 to 3 and 4 to 7 substrings are equal and they are lps. So,
 $Pi[7] = Pi[6] + 1 = 3 + 1 = 4$ and problem is solved

Preprocessing Phase: Construction of Pi Table

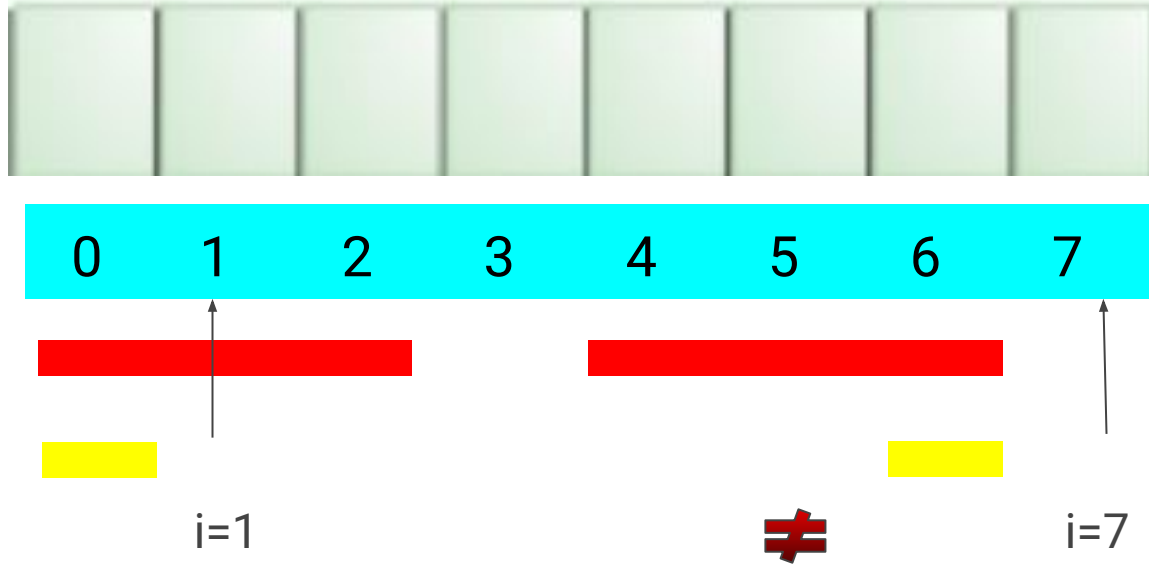


Preprocessing Phase: Construction of Pi Table



If they are not equal, then dig deeper. Yellow bar indicates lps of 0 to 2 or 4 to 6 where first yellow bar is prefix part of lps of 0 to 2 and second yellow is suffix part

Preprocessing Phase: Construction of Pi Table



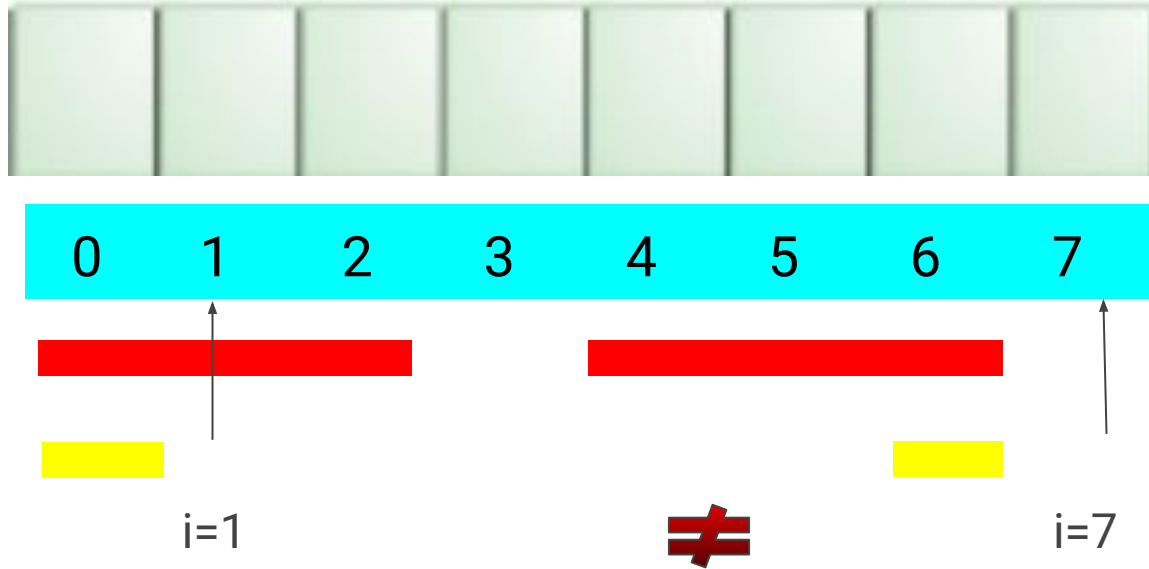
of LPS of string from 4 to 6

Preprocessing Phase: Construction of Pi Table



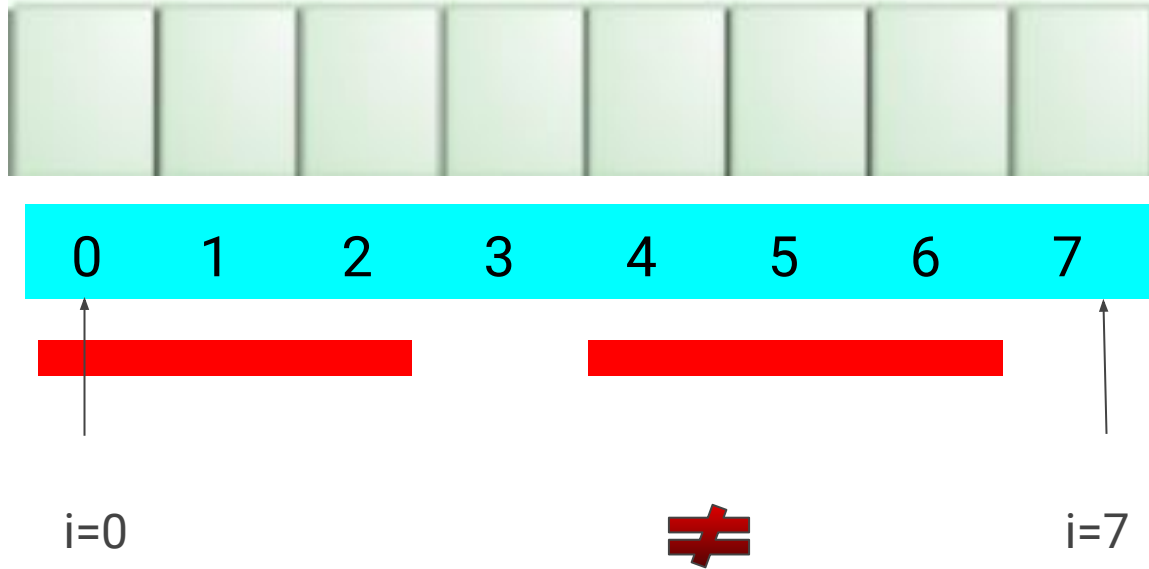
Repeat the same process. If equal, $Pi[7]=Pi[1]+1=1+1=2$. Else,

Preprocessing Phase: Construction of Pi Table



We have to **dig deeper**.

Preprocessing Phase: Construction of Pi Table



This is the **base case** and we cannot move from here. If these 2 characters are equal, we get $Pi[7]=Pi[0]+1=0+1=1$. Else $Pi[7]=Pi[0]=0$

Pseudo code of the algorithm

HashMap<int,int>dp;String s;

Int Pi_fill(int index){// **computes LPS of substring ranging from 0 to index**

if(dp.containsKey(index)) return dp[index];// **memoisation step**

Int temp=Pi_fill(index-1);// **starting from LPS of prefix ending at index-1**

while(temp!=0&& s[temp]!=s[index]) temp=Pi_fill(temp-1);// **digging until match comes**

if(temp==0&& s[temp]!=s[index]) return dp[index]=temp;// **base case**

Else return dp[index]=temp+1;// **match occurred so, LPS will be 1+index of that character**



Text processing

j

.....

a a b x a a b x c a a b x a a b x a y

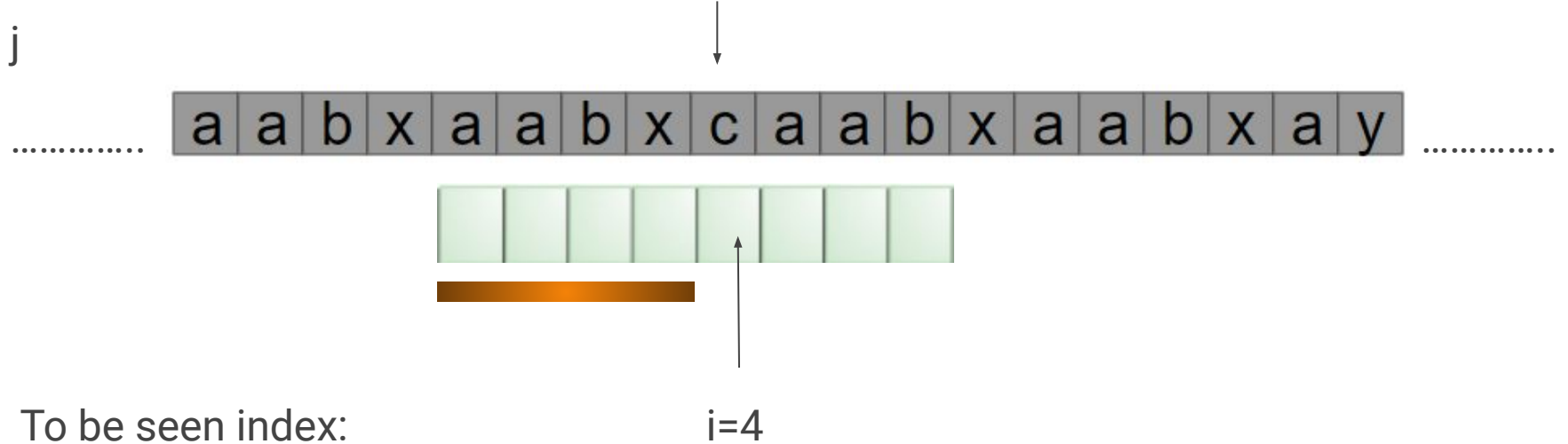


i=7

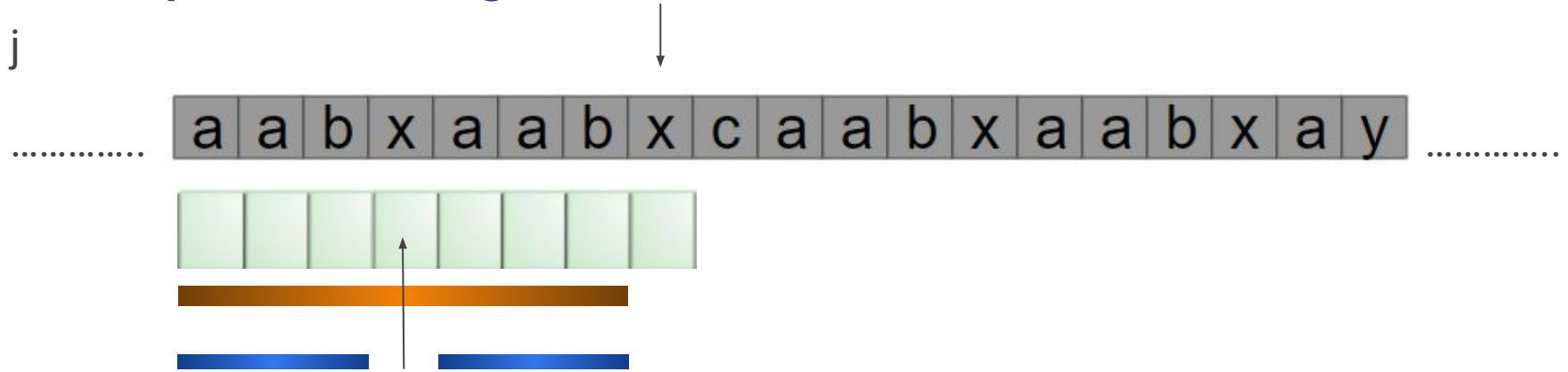
Mismatch index:



Text processing



Text processing



Blue bars indicate lps of the substring of pattern from index 0 to index 6

Text processing

j

..... a a b x a a b x c a a b x a a b x a y



mismatched index: i=6

i=6



Text processing

j

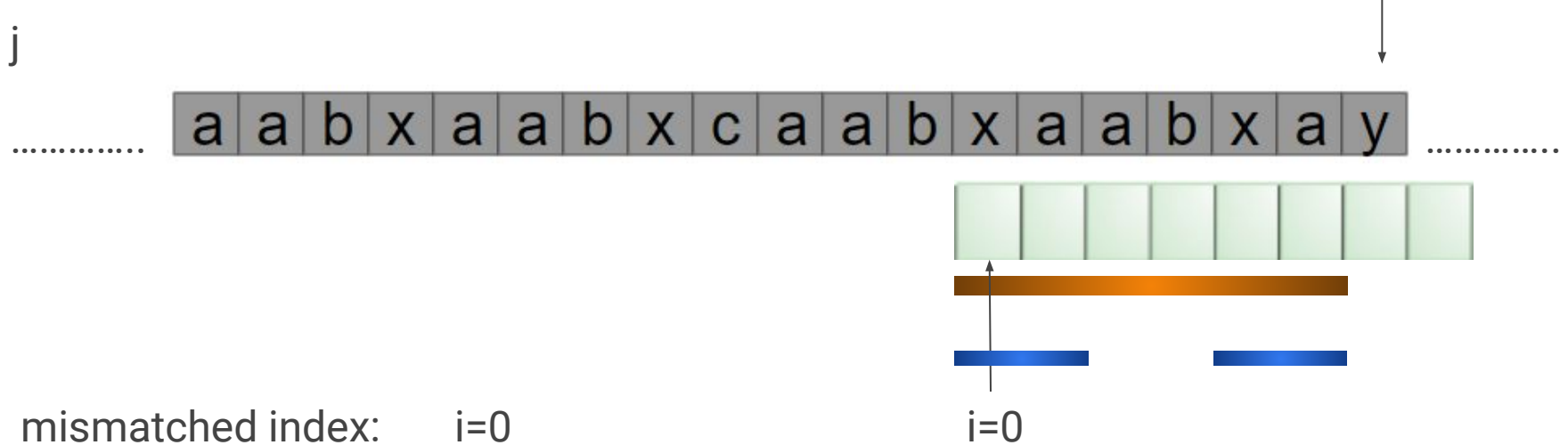
..... a a b x a a b x c a a b x a a b x a y



mismatched index: $i=2$

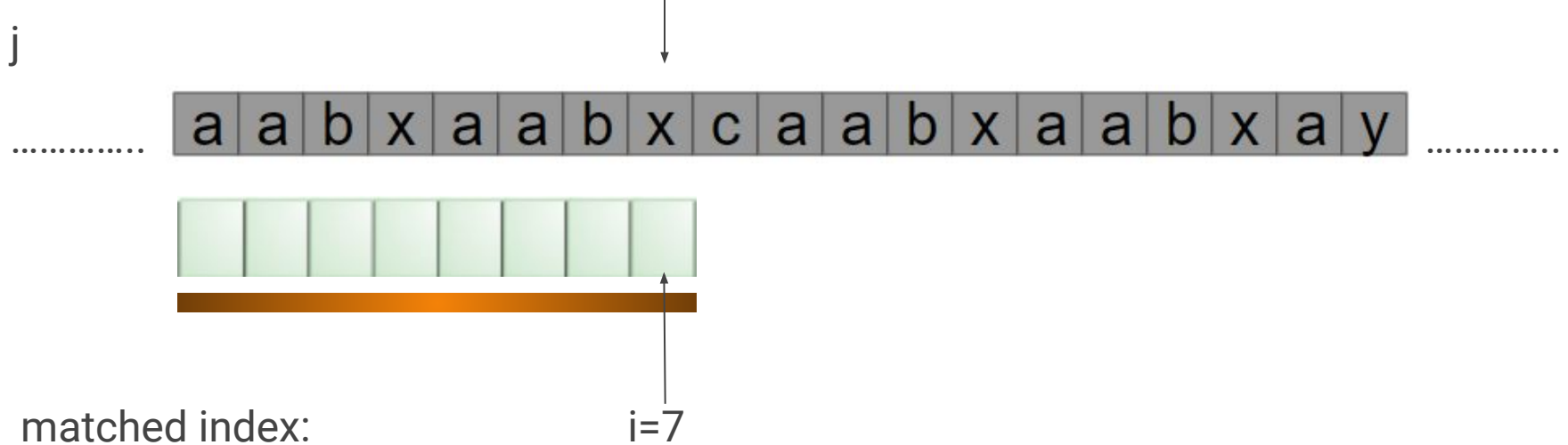


Text processing



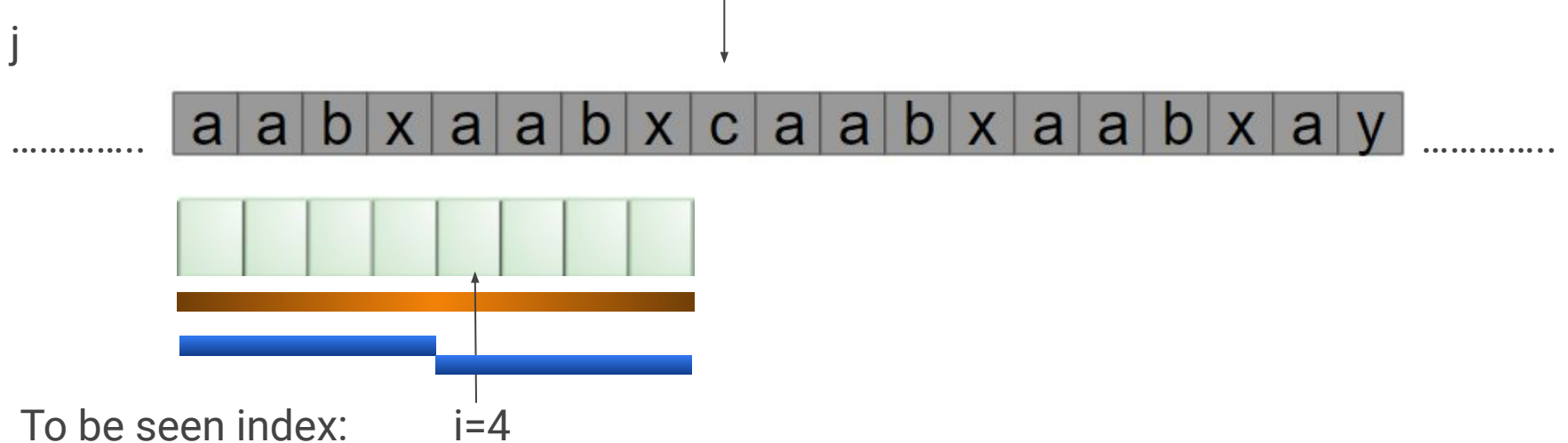
Can't move still more deeper, but inequality is still there. So , we have to move ahead j pointer and continue the search afresh from $j+1$ and $i=0$.

Text processing-whole pattern match case



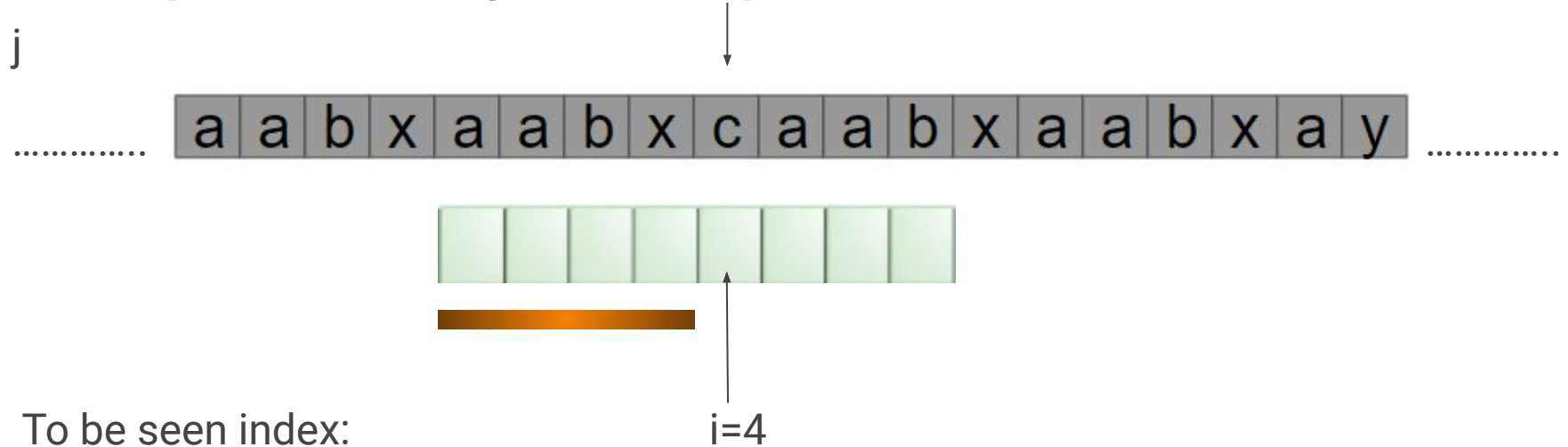
So, we will push "j-pattern.length()" into the final answer set. So, what next ?

Text processing-whole pattern match case



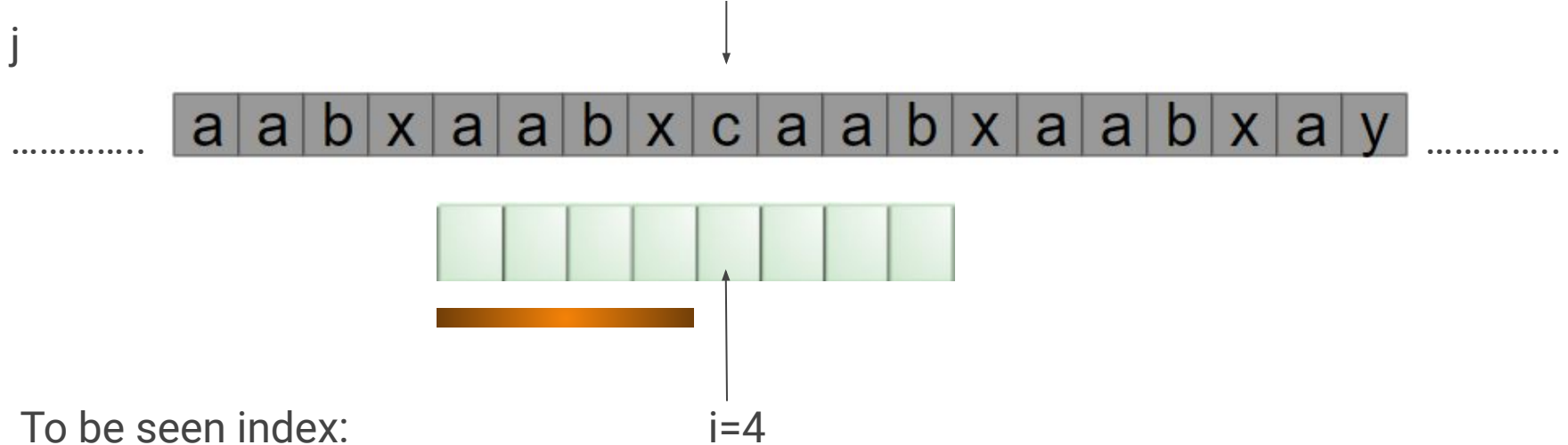
We can bring i pointer to LPS of the whole string, before searching. This works because, on moving to the next window we remove one character only at front and add one more character at back. So, LPS of whole string is still intact in its suffix realisation (as it is proper suffix, it doesn't have first character) and it is largest part

Text processing-whole pattern match case



to be same and we can ignore some windows between the old and new windows because, there is gonna be mismatch at some index, which lies within the range of old window itself. Otherwise, LPS of the pattern will be wrong. So, here is the reason for this algorithm's non-redundancy as it makes use of the events occurred till now.

Text processing-whole pattern match case



To be seen index:

So, text pointer j never returns back or backtracks. So, time complexity of the algorithm is $O(m+n)$ where m is pattern's length and n is text's length. $O(m)$ is due to pre-processing complexity and $O(n)$ is due to text processing.

Pseudo code

```
Int j=0; set<int>answer;  
void KMP(text,pattern,int i){  
    j++;  
    if(j==text.length+1) return;  
    if(i<pattern.length-1&&text[j-1]==pattern[i]) KMP(text,pattern,i+1);  
    if(i==pattern.length-1&&text[j]==pattern[i]){  
        answer.add(j-i-1);  
        KMP(text,pattern,Pi[i]);  
    }  
    Else if(i==0) KMP(text,pattern,i)  
    Else{  
        Int k=Pi[i-1];  
        while(k>0&&pattern[k]!=text[j-1]) k=Pi[k-1];  
        if(k==0&&pattern[k]!=text[j-1]) KMP(text,pattern,0);  
        Else KMP(text,pattern,k+1)  
    }  
}
```

Every time, we repeat the algorithm, text pointer j is always incremented by 1

Matching case: If matched, then move further. If the whole string matches, add starting index of the window and backtrack i pointer to the Lps of the whole string and repeat the algorithm

Mismatch case: Backtrack the i pointer to the LPS of the matching prefix before i and again check for match and repeat this process until match is obtained. If match exists, update i and repeat the algorithm. Else, repeat the algorithm with i=0


Pattern Matching with Automata

A top up on the previous KMP algorithm

Terms and meanings

- Traversal cases(Transitions in DFA, except for the ones corresponding to match): Key value pair of every unequal character in the text and the position to go if the corresponding key character is present instead of the one present in the string .For example for the string ABABA, the traversal cases for the 4th character B (index 3) is $\langle a, 1 \rangle$.This means that if every thing got matched till third character, and 4th character is not matched, then if the encountered mismatched character in text is “a” then we can directly move to index 1(2nd character) otherwise, we have to move to the 1st character of the string and repeat the algorithm.These represent transitions in DFA machine and in the algorithm these are represented by a hashmap.
- **Note:** *One more character which is not used in the text is appended to the pattern at last for indicating the final state of DFA machine*

Brief Overview

- As we all saw, the previous KMP algorithm involved backtracking of pattern pointer during text processing. So, what if we have a bit more spoonfed version of KMP algorithm?
 - Like, we keep on digging in until we get a character which succeeds LPS is equal to the mismatched character in the text. But, if we have a guide book for every mismatch regarding where to go?, depending on the unequal character in the text. In KMP, we look for only equality or inequality, whereas in Automata approach, we value inequality and move to other states in one step itself.
- 

How is it a top up on KMP?

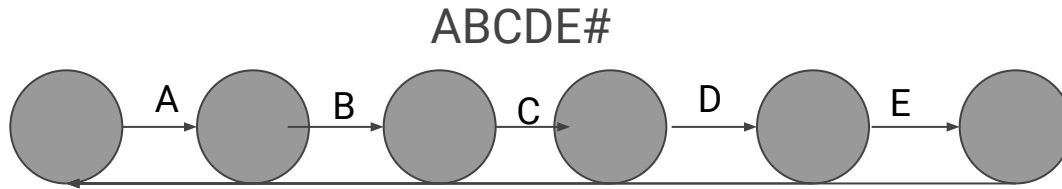
- We have to push back an unused character in the text at end, suggesting the final state of the state of the machine.
- First like, KMP we need to prepare Pi table.
- Using Pi table, we prepare traversal cases for characters in the string or transitions of the state transition machine.
- Text processing will be spoonfed



Logic for transition graph construction

- A State transition graph tells us to which state we are supposed to go if a particular input is obtained.
- Case 1: If there is a match it means that it corresponds to a transition which move our pattern pointer to the next pointer.
- Case 2: If there is a mismatch, then we have to go back depending on which character actually came, So, using the concept of LPS, we can say that again we have to go to succeeding character of lps of prefix ending at one character before the current character and if they are equal then we will again move pattern pointer to that position and continue. Otherwise, we again obey the transitions except for the ones corresponding to case 1 or (traversal cases) because we again repeat the same process. So, this is a sought of dp where they might be repeated subproblems

Realisation of DFA from pattern string



Where # indicates final state.

Pseudo code for graph preparation

```
Hash_map Pi  
vector<HashMap> Graph(string pat){  
    Pi_fill(pat);  
    pat.push_back('^');
```

```
vector<Hash_map> Graph(m+1);
```

Filling Pi table and pushing an exceptional character into pattern for detecting final state

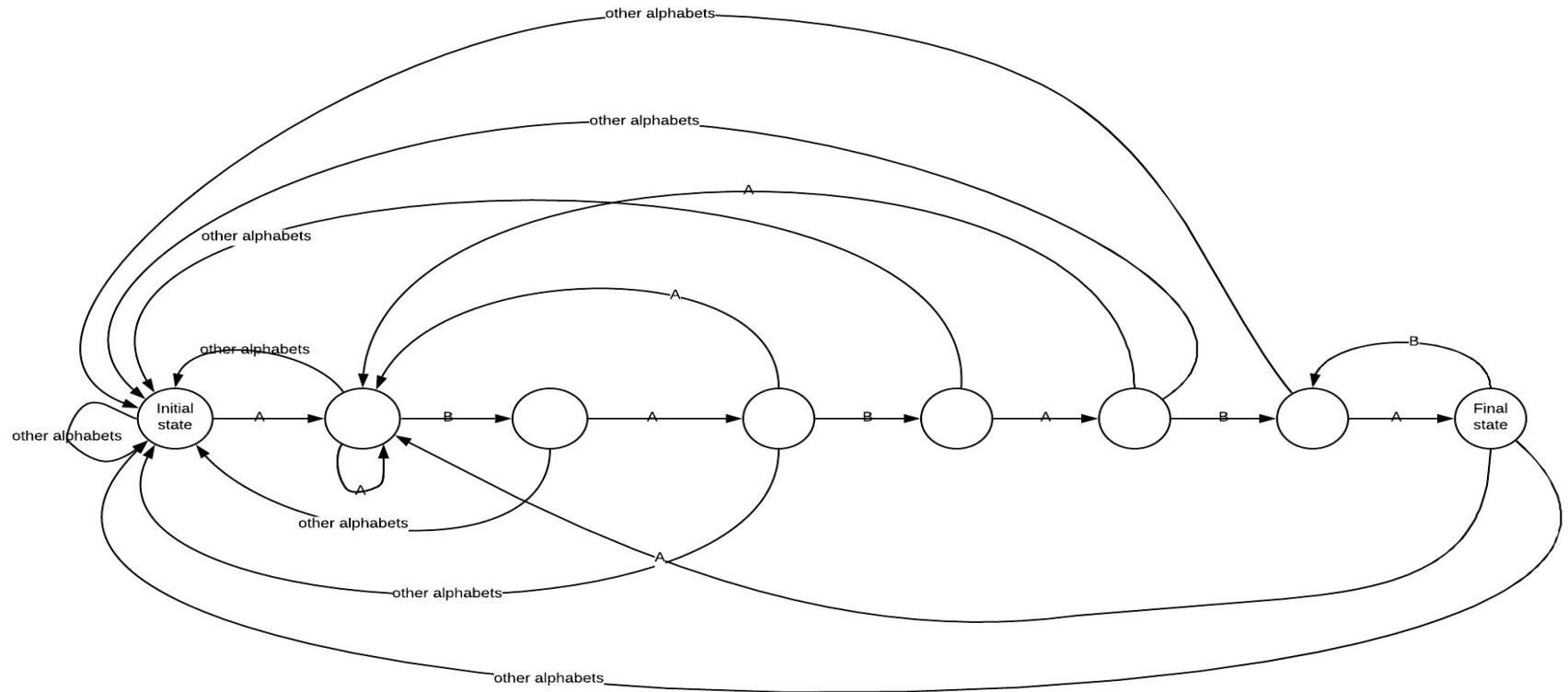
Graph or traversal cases as vector of hashmaps for knowing where to go if mismatch occurs. If there is no position for the unequal mismatched character, that means we have to go to initial state again



Pseudo code for graph preparation

```
for(int pointer=1;pointer<=m;pointer++){  
    Hash_map temp=Graph[Pi[pointer-1]];  
    Graph[pointer]=temp;  
    if(Graph[pointer].find(pat[pointer])!=Graph[pointer].end())  
        Graph[pointer].erase(pat[pointer]);  
    if(pat[Pi[pointer-1]]!=pat[pointer])  
        Graph[pointer][pat[Pi[pointer-1]]]=Pi[pointer-1]+1;  
}  
return Graph;  
}
```

A Bottom-up implementation for preparing graph. We want to find all traversal cases for a particular character in a string, we keep on digging into LPS of the string and again into its LPS and so on to know where all we can go to from that state if mismatch occurs. So for knowing traversal cases for a particular index, we should know all traversal cases of all indices before it. So we start from 1st index and first index has null set of traversal cases because it can't move away from that state unless a match occurs and from pointer=1 we add all traversal cases of LPS of prefixes ending at index pointer-1 and include the case of mismatch of Pi[pointer-1]+1 and exclude the case when key character and pat[pointer] are same as dfa can't have 2 transitions with same input



Example DFA for checking occurrences of the pattern “ABABABA” in the text

A	B	A	B	A	B	A	#
Φ	<A-1>	Φ	<A-1>	Φ	<A-1>	Φ	<A-1> <B-6>

Graph of the pattern “ABABABA” obtained in the algorithm to represent traversal cases of all characters of the pattern. This means that if mismatch is obtained at index $i=4$ then we have to move i to 0 . And if a mismatch is obtained at $i=5$ and if that mismatched character in text is A then, we have to move i to 1 and if any other character is encountered in the text, we have to move i to 0. This coupled with pattern string, represents the entire DFA. If a match is obtained at $i=0$ then, we increment i to 1 similarly if it is found at $i=1$, we move i to 2 and so on. So we get an entire elaborate guide book which is exactly the same as an automata machine. ‘#’ represents the final state. If i points to ‘#’, this means that in automata, we have reached the final state and we have to move on from there to process the further text and find other occurrences as well. This will definitely happen because we’re assuming that ‘#’ doesn’t occur in the text so a mismatch is bound to occur and it follows the instructions present in the set of traversal cases of ‘#’ present in graph of the pattern.

Complexity analysis

- For graph preparation we need $O(\text{no_char} * m)$ time as at max a hash map may have all characters except itself. So, copying complexity is $O(\text{no_char})$ and there are $m+1$ nodes. So is the complexity.



Manacher's Algorithm

Finding Longest Palindromic Substring

Brief Overview

- Manacher's algorithm finds the longest palindromic substring in string using one of the famous and most used programming paradigm **Dynamic Programming** in **time complexity of $O(n)$** .
- Palindromes are used at quite a many places in real life implementing **compression algorithms**. There are researches about biological sequence compression algorithms.
- This algorithm exploits the following fact for a palindrome:
 - **Palindromes are symmetric** around its center within the boundary.



Naive Approach

In order to find a palindrome in a string we **expand in both directions considering** each element as center and store the length of palindrome in an array say P. Then we **traverse the array P and find the maximum value $P[k]$** representing the length of longest palindromic substring having center as k and length $P[k]$.

For ex: let T: “#\$a\$b\$a\$b\$a\$b\$a\$@”

It has P: 00103050705030100

This is the naive approach which take **$O(n^2)$ time complexity.**



But... we have some clever insights which the Manacher's Algorithm uses, from the properties of a palindromic string with the help of which we can lower down the complexity from $O(n^2) \rightarrow O(n)$.

Right now what is hurting us is the expansion around each center. What if **we cut down the number of expansions** and calculate the length using the **values calculated earlier**.

Let's see how..



Firstly let us realise that each palindrome has a **palindrome center** and that a palindrome is symmetric around its center **within its boundary** and not out of it. For example:

“abababa” has a palindrome **center at index 3** (0-indexed) ,

“\$a\$b\$a\$” has a palindrome **center at index 3** (0-indexed) ,

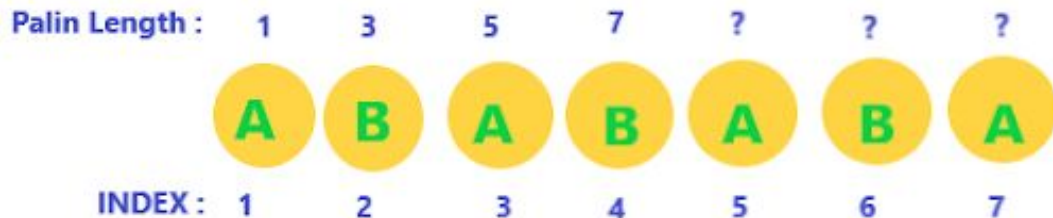
Even **“aa”** has a palindrome center, but that's right in middle of the two characters. **In order to avoid such situation** and to have a palindrome center such that all the palindromes in the string are explored we will **modify the string** and insert special characters (for the sake of this article) **#,\$ and @**.

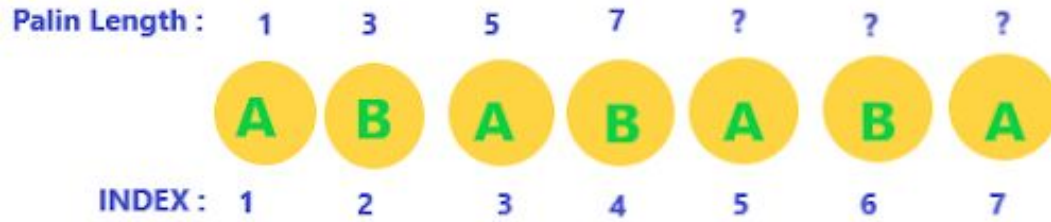
Hence for example any string “ababa” will be modified to

T: “#\$a\$b\$a\$b\$a\$@”

Terms and Definitions

- **Center (C):** It's the center of the palindromic string which has been explored the rightmost yet. For example index 4 here.
- **Right Boundary (r):** It is the right boundary of the palindromic string which has been explored the rightmost yet for example index 7 here
- **Mirror Center:** It is the character at distance equal to the currently processing character from the Center C to the left of C.
- **P array:** Array storing the palindromes length with i as center and having length P[i].





Since a palindrome is **symmetric about its center and within boundary**, Palin Length at :

5 equal to 5

6 equal to 3

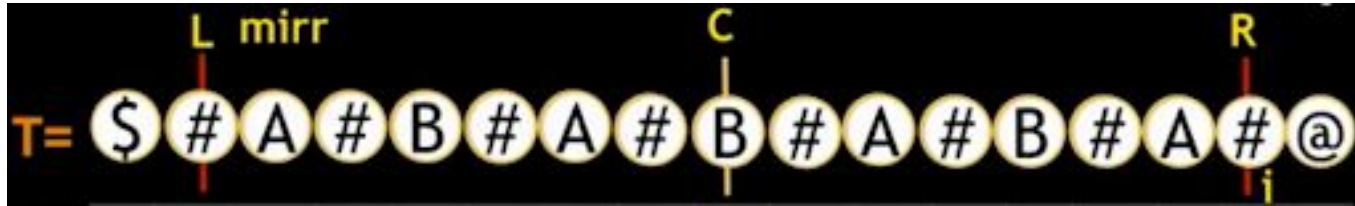
7 equal to 1

- Manacher's Algorithm exploits this idea quite cleverly.
 - It says that if we have **already calculated** the palindromes length around each center **before the center C**, we are not needed to explicitly calculate the palindrome length of characters **in range of the farthest window**.
-
- So let's maintain the pointers
 - **C**: current center
 - **r**: right-most point
 - **i**: current element to be explored



Now, let's look at the implementation...

We will calculate the **P array** discussed in the earlier section from which the **longest palindrome can easily be calculated** in a **single traversal** of the array. So let's see how to calculate P array for the following string:




Tricky Point..

*If the **answer** of current character as mirror length **exceeds the length of current** consideration of palindromic string then we'll have to **limit** our answer for that character to the **right boundary** of the currently considered palindromic string.*

If the answer of current character as mirror length is within limit then we consider that as the current answer.

Now we **expand** the palindromic length for current character in **both directions** from the current palindromic length.

Then we update the mirror position to the position of current character, if the palindromic length of current character exceeds the right boundary of the current palindrome into consideration.



l:



\$ # A # B # A # B # A # B # A # @

P:

0

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

P:

0

0

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

P:

0

0

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

P:

0

0

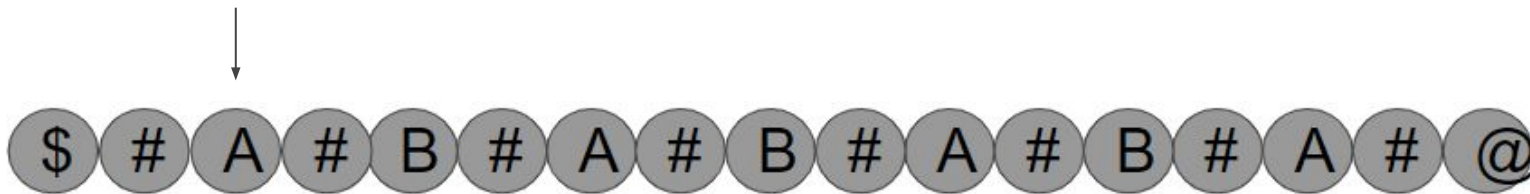
C:



r:



I:



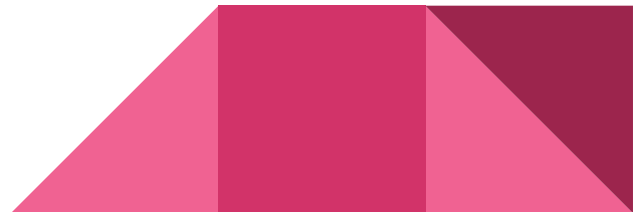
P:

0 0 1

C:



r:



I:



\$ # A # B # A # B # A # B # A # @

P:

0 0 1

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0

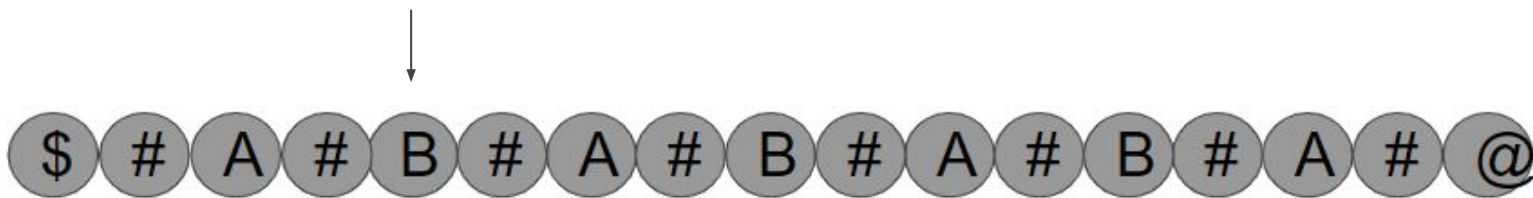
C:



r:



l:



P:

0 0 1 0

C:



r:



I:



\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3

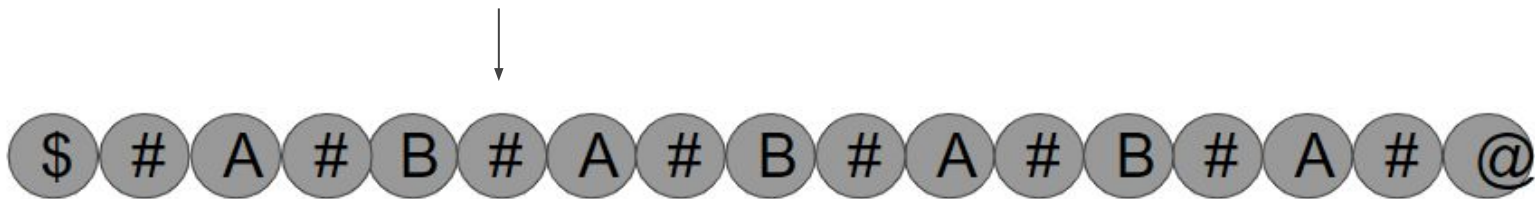
C:



r:



l:



P:

0 0 1 0 3 0

C:



r:



l:



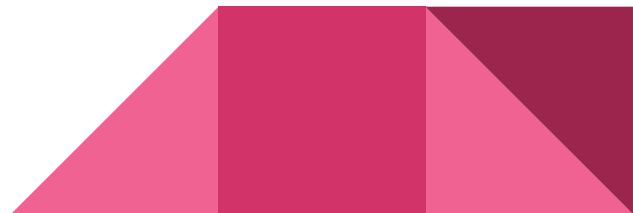
P:

0 0 1 0 3 0

C:



r:



l:



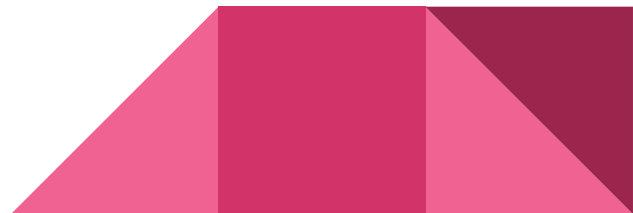
P:

0 0 1 0 3 0

C:



r:



I:



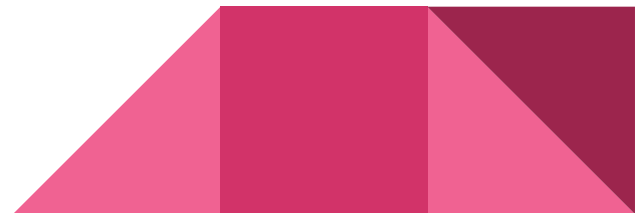
P:

0 0 1 0 3 0 5

C:



r:



l:



P:

0 0 1 0 3 0 5

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0

C:



r:



I:



\$ # A # B # A # B # A # B # A # @

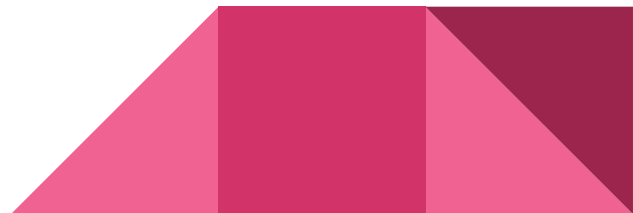
P:

0 0 1 0 3 0 5 0

C:



r:



I:



\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7

C:



r:



I:



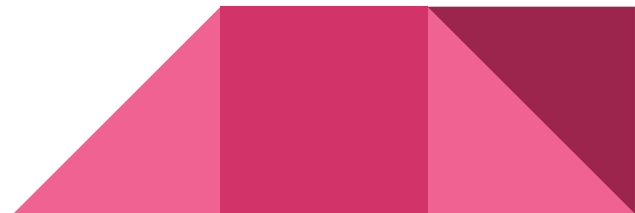
P:

0 0 1 0 3 0 5 0 7

C:



r:



l:



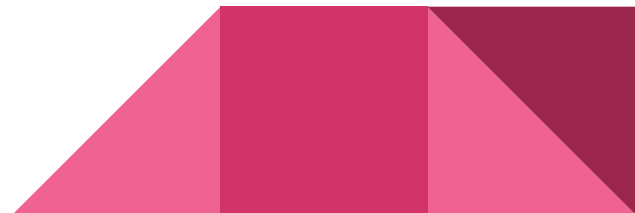
P:

0 0 1 0 3 0 5 0 7 0

C:



r:



l:

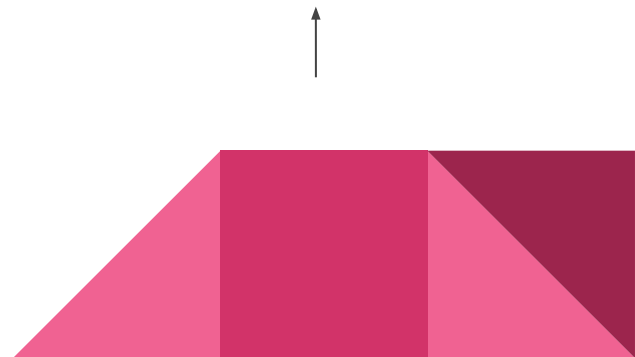
\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7 0

C:

r:



l:

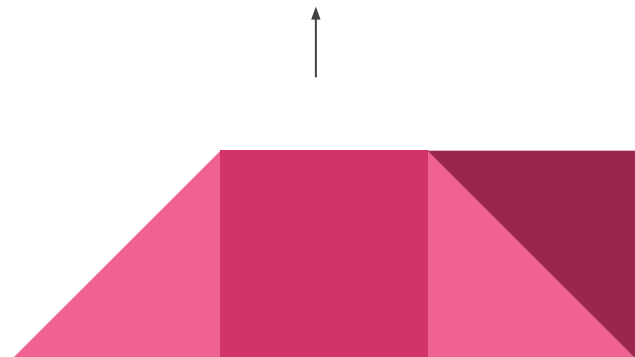
\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7 0 5

C:

r:



l:

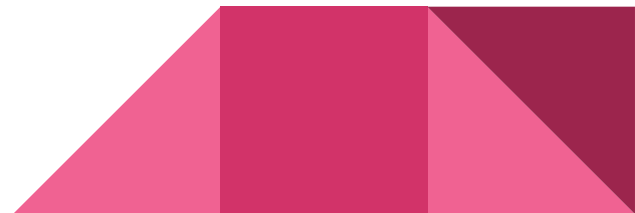
\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7 0 5

C:

r:



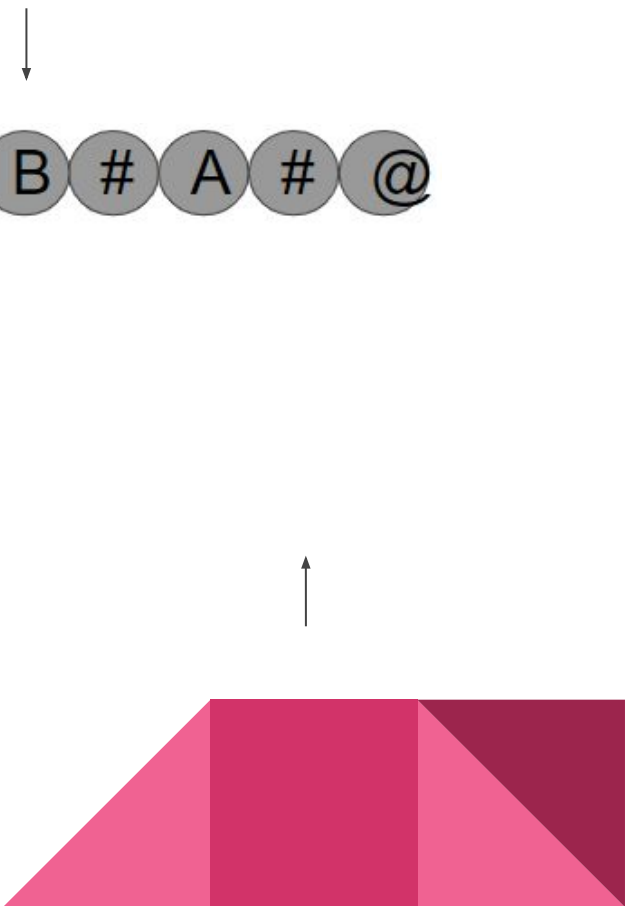
l:

\$ # A # B # A # B # A # B # A # @

P: 0 0 1 0 3 0 5 0 7 0 5 0

C:

r:



l:

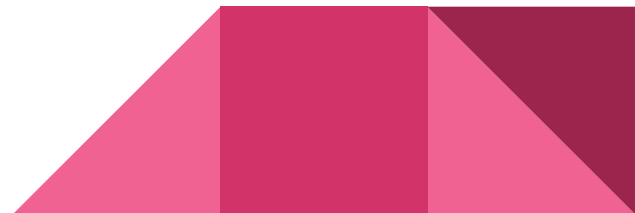
\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7 0 5 0 3

C:

r:



l:

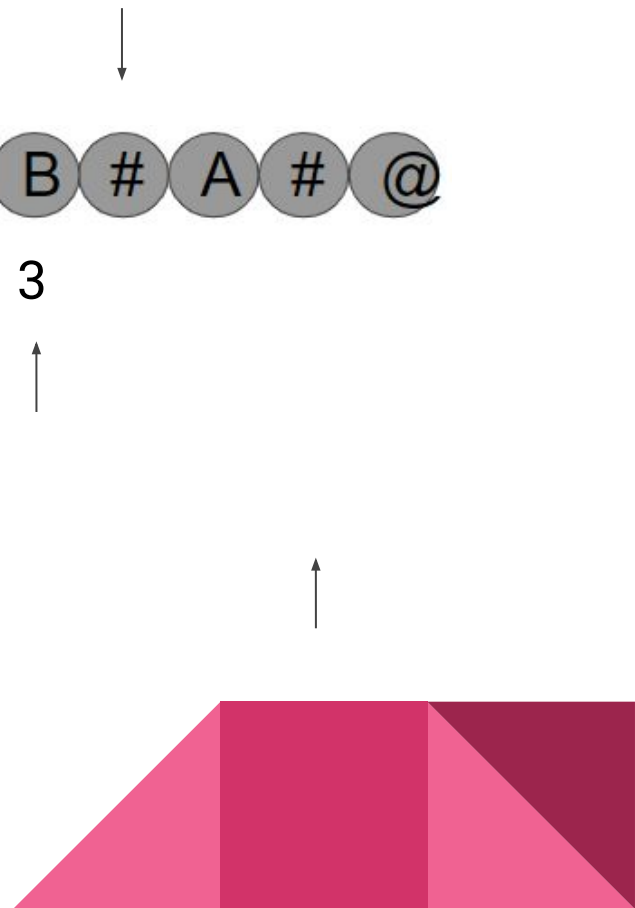
\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7 0 5 0 3

C:

r:



l:



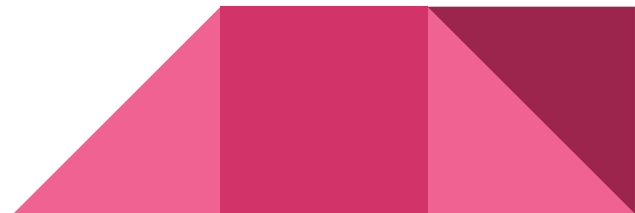
P:

0 0 1 0 3 0 5 0 7 0 5 0 3 0

C:



r:



l:

\$ # A # B # A # B # A # B # A # @

P:

0 0 1 0 3 0 5 0 7 0 5 0 3 0

C:

r:



I:



\$ # A # B # A # B # A # B # A # @

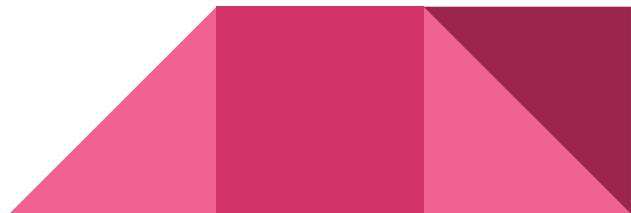
P:

0 0 1 0 3 0 5 0 7 0 5 0 3 0 1

C:



r:



l:



\$ # A # B # A # B # A # B # A # @

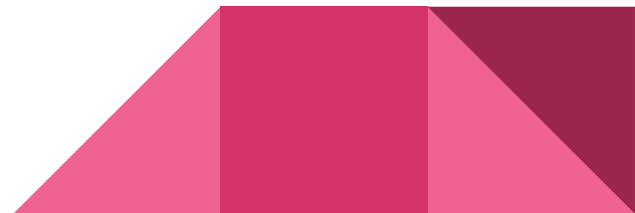
P:

0 0 1 0 3 0 5 0 7 0 5 0 3 0 1

C:



r:



I:



\$ # A # B # A # B # A # B # A # @

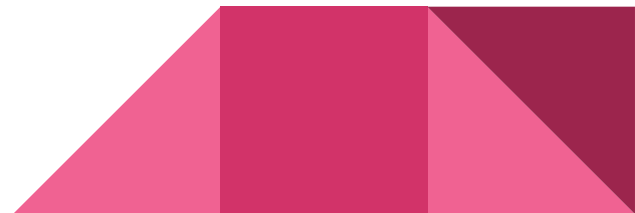
P:

0 0 1 0 3 0 5 0 7 0 5 0 3 0 1 0 0

C:



r:



And That's how we get the P array..

Now let's see its code implementation...



Code Implementation

```
1  vector<int> lpas(string T)
2  {
3      int n=T.length();
4      vector<int> P(n,0);
5      int C=0, R=0;
6      for(int i=1;i<n-1;i++)
7      {
8          int mirr = 2*C - i;
9          if (i < R)
10             P[i] = min(R-i, P[mirr]);
11          while(T[i+(1+P[i])] == T[i-(1+P[i])])
12             P[i]++;
13          if (i + P[i] > R)
14          {
15              C= i;
16              R=i+P[i];
17          }
18      }
19      return P;
20 }
```

Time Analysis

We observe that we search for the longest palindrome in **linear traversal of the string**. Moreover once the right boundary is increased, it's never reduced. So the **running time complexity of this algorithm is $O(n)$** .

END



Z-Algorithm

Linear time pattern matching algorithm

Brief Overview

- Z-Algorithm is used to **find** all the occurrences of a **pattern** as a substring in a string in a linear traversal of the string, i.e. **$O(n)$ time complexity** using a well-known and applied programming paradigm, **Dynamic Programming**.
- Although KMP algorithm also performs the same task but I want you to focus on the below points:
 - Z-Algorithm is much easy to implement than the KMP Algorithm.
 - LPS array calculated in KMP is a general and quite a used concept applied in many applications but Z array is specifically calculated to perform this task.
- Pattern matching has application such as information retrieval, virus scanning, **DNA sequence analysis**, data mining, network security and was a crucial algorithm for the **Human Genome Project (1990 – 2003)**.

Naive Approach

- In a brute way we will **check for each character** whether the pattern is matching from that character onwards or not.
- If it is then we'll print its index otherwise we'll move forward and check for the next character.
- For the string "aabcbabcd" and pattern "abc"

Let's see how...



a a abcbabcd	a bc		aabcb <u>a</u> abcd	<u>a</u> bc	✗
a a <u>a</u> abcbabcd	a <u>a</u> bc	✗	aabcb a abcd	a bc	
a a abcbabcd	a bc		aabcb a bcd	a bc	
a a a bcbabcd	a a bc		aabcb a a bcd	a a bc	✓
a a a a bcbabcd	a a a bc	✓	aabcbab <u>a</u> bcd	<u>a</u> bc	✗
aab <u>a</u> cbabcd	<u>a</u> bc	✗	aabcbab <u>a</u> cd	<u>a</u> bc	✗
aab <u>a</u> cbabcd	<u>a</u> bc	✗	aabcbab <u>a</u> cd	<u>a</u> bc	✗

Analysis of naive approach

We realise that **brute approach takes $O(n^2)$** time complexity in the worst case but **can we reduce this to $O(n)$** time complexity using some preprocessing ?


The answer is **YES !!!**

This can be achieved using **Z-Algorithm** which creates the **Z-array** as a **preprocessing step**.

Firstly we'll see **what is Z-Array** and **how to find all the matches** in the string using the Z-Array.



What is Z-Array ?

- Let the Z-Array be defined with symbol Z. The i^{th} term i.e. **Z[i]** represents the **length of longest substring starting from i^{th} index which is also the prefix of the string.**
 - For ex. **“abacabfab”** has Z array as:
 - Z: **901020020**
 - Note: Z[0] is always equal to the length of the string but for calculation purpose we take Z[0]=0.
 - Once if we know how to calculate Z-Array in $O(n)$, **can we use it to find all possible positions where a pattern matches in a string ?**
 - **YES!! We can....**
- 

Going from Z-Array to Pattern Matching

Let Pattern to be found be P in the string S .

We will concatenate P and S along with a **special character**(not present in the string) '\$' to form

T : " $P\$S$ "

And find the Z array for T . **Once we get it, we just have to look for all those indices i where $Z[i]$ is equal to length of pattern.** This i^{th} index is the starting point of the matched pattern in the string T .

To get **index in S** , simply subtract $(\text{len}(P)+1)$ from i .




Terms and Definitions

i: current pointer to the character whose $Z[i]$ is to be calculated

Z: Array where $Z[i]$ represents the length of the longest string starting at i^{th} element which is also the prefix of the string.

L: It points to the left boundary of the rightmost Z-box explored so far.

r: It points to the right boundary of the rightmost Z-box explored so far.



Calculating Z-Array for a string

We will calculate it for the string....

a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



i: ↓

a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

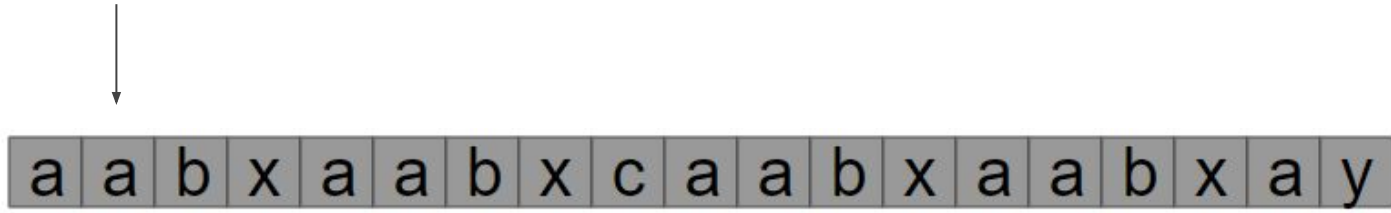
z: 0

L: ↑

r: ↑



i:



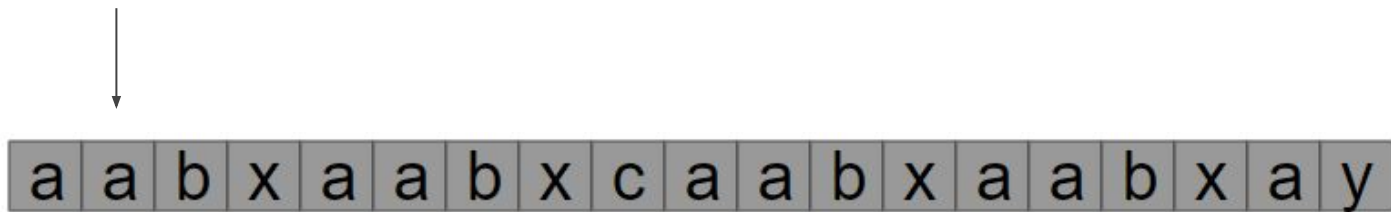
Z: 0

L:

r:



i:



Z: 0 1

L:

r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4

L:



r:



i:



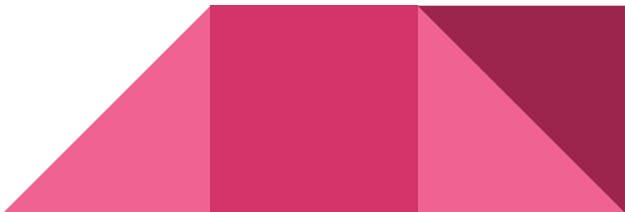
a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4

L:



r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4 1

L:



r:



i:

a a b x a a b x c a a b x a a b x a y

Z: 0 1 0 0 4 1 0

L:

r:



i:

a a b x a a b x c a a b x a a b x a y

Z: 0 1 0 0 4 1 0 0

L:

r:



i:

a a b x a a b x c a a b x a a b x a y

Z: 0 1 0 0 4 1 0 0

L:

r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4 1 0 0 0

L:



r:



i:

a a b x a a b x c a a b x a a b x a y

Z: 0 1 0 0 4 1 0 0 0

L:

r:



i:



a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4 1 0 0 0 8



L:



r:



i:



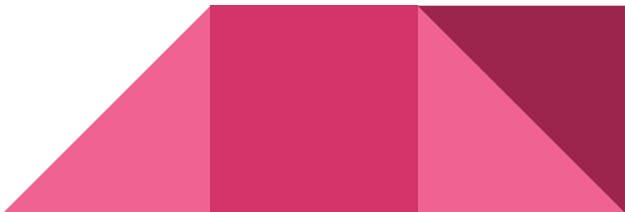
a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4 1 0 0 0 8



L:

r:



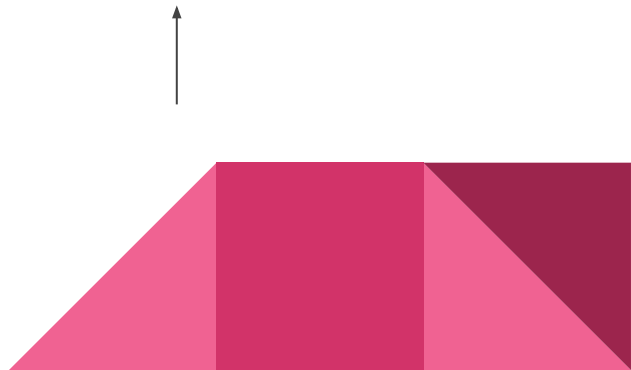
i:

a a b x a a b x c a a b x a a b x a y

Z: 0 1 0 0 4 1 0 0 0 8 1

L:

r:



i:

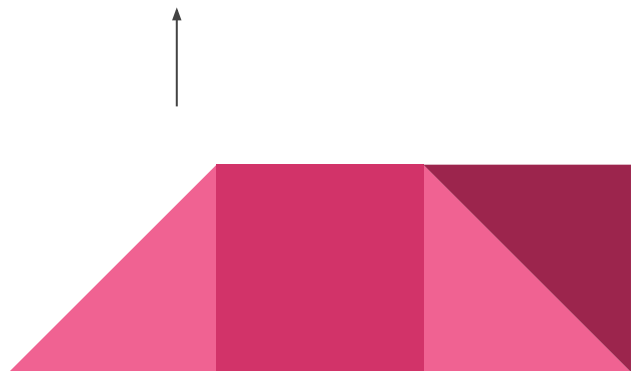
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0

L:

r:



i:

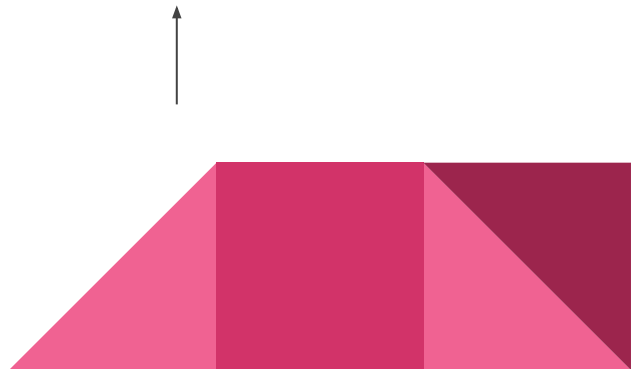
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0

L:

r:



i:

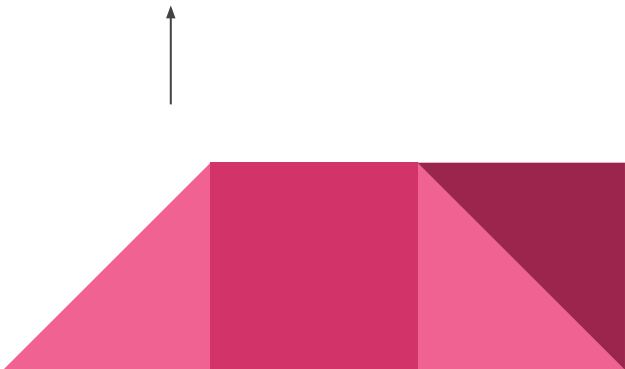
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 4

L:

r:



i:

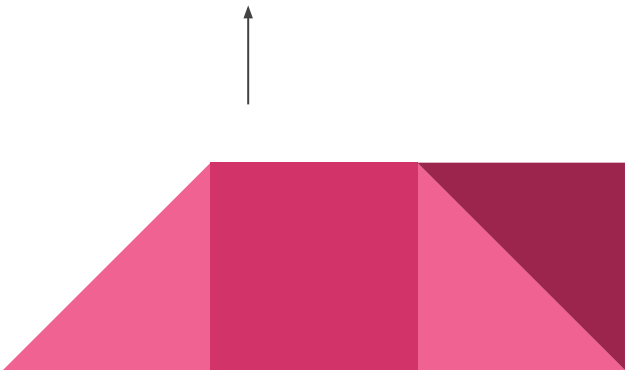
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5

L:

r:



i:

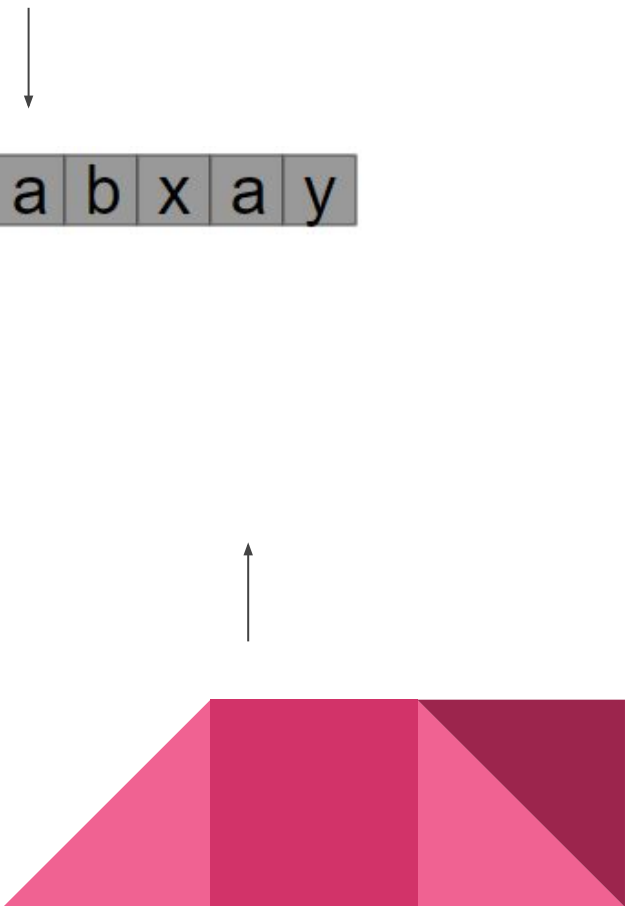
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5

L:

r:



i:

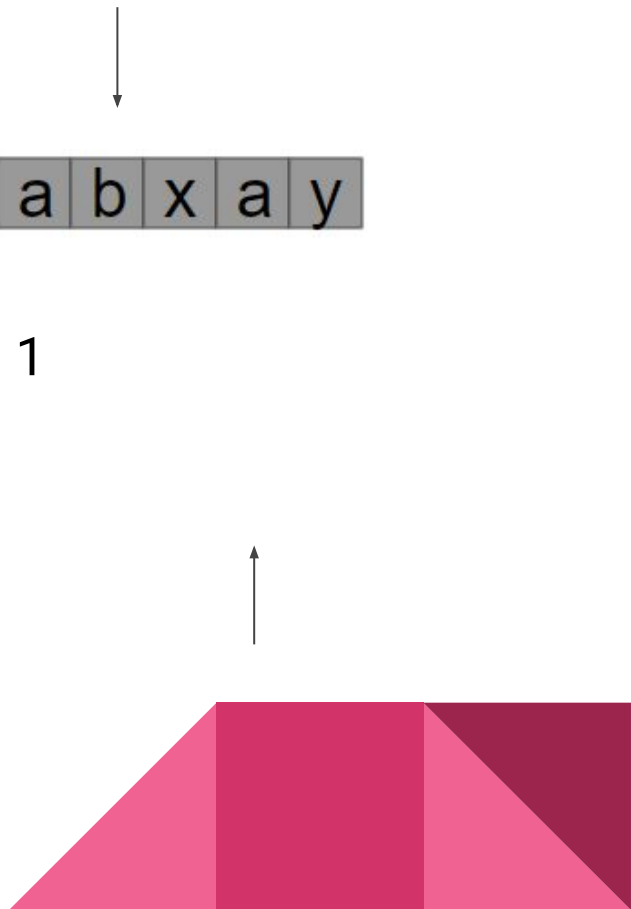
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1

L:

r:



i:

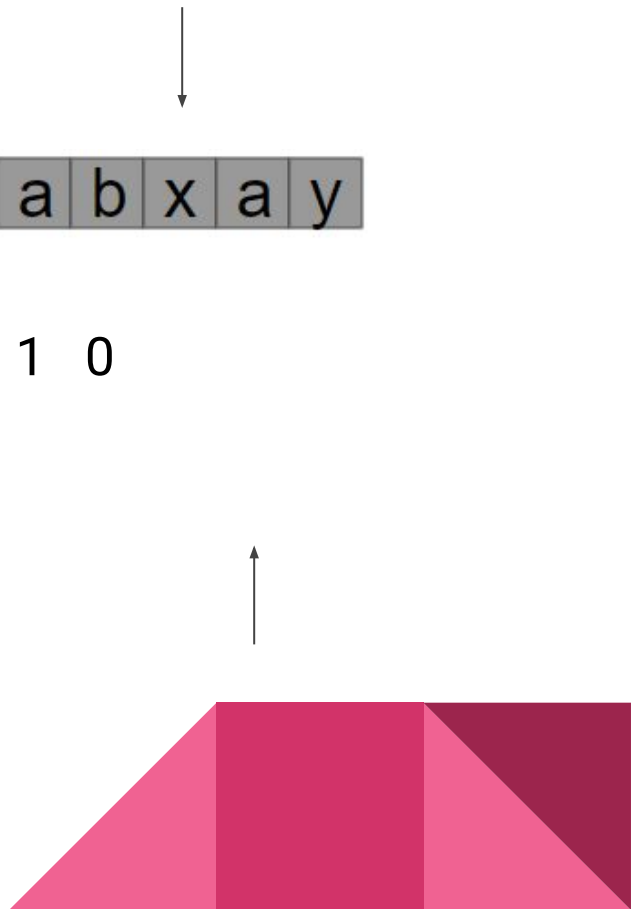
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0

L:

r:



i:

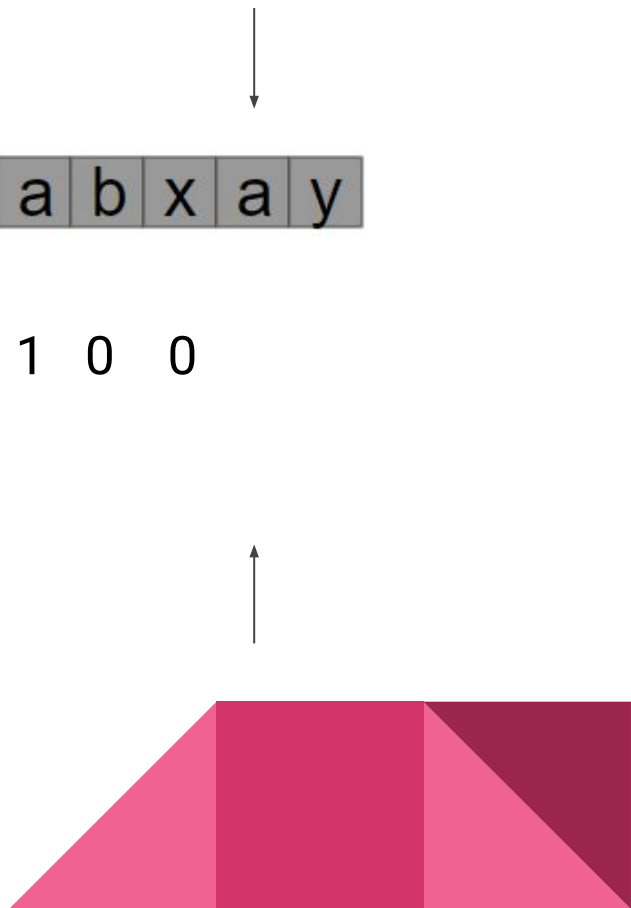
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0

L:

r:



i:

a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0

L:

r:



i:

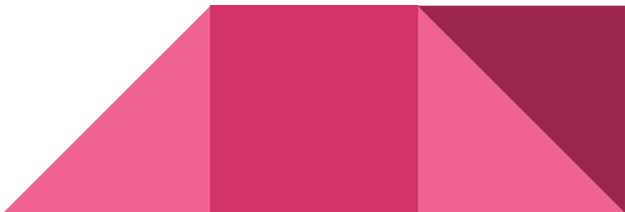
a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0 1

L:

r:



i:

a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



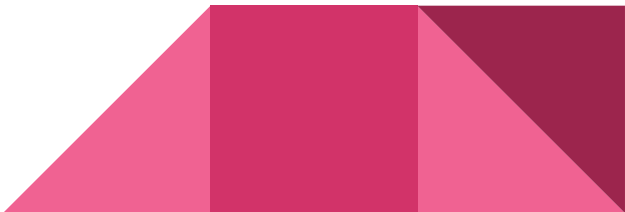
Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0 1

L:



r:



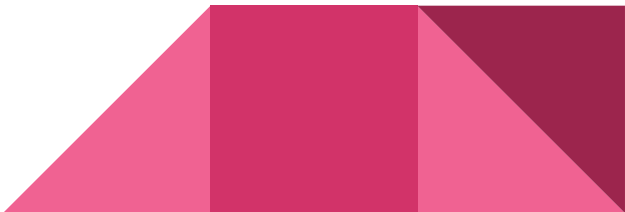
i:

a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0 1

L:

r:



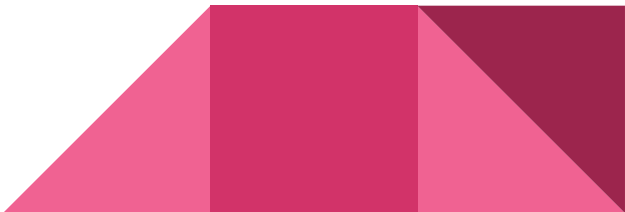
i:

a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Z: 0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0 1 0

L:

r:



i:

a a b x a a b x c a a b x a a b x a y

Z:

0 1 0 0 4 1 0 0 0 8 1 0 0 5 1 0 0 1 0

L:

r:

Z-Array



And That's how we get the Z array..

Now let's see its code implementation...



Code Implementation

```
1  vector<int> getZArr(string s)
2  {
3      int n=s.length();
4      vector<int> z(n,0);
5      int L = 0, R = 0;
6      for (int i = 1; i < n; i++)
7      {
8          if (i > R)
9          {
10             L = R = i;
11             while (R < n && s[R-L] == s[R])
12             {
13                 R++;
14             }
15             z[i] = R-L;
16             R--;
17         }
```

```
18     else
19     {
20         int k = i-L;
21         if (z[k] < R-i+1)
22         {
23             z[i] = z[k];
24         }
25         else
26         {
27             L = i;
28             while (R < n && s[R-L] == s[R])
29             {
30                 R++;
31             }
32             z[i] = R-L;
33             R--;
34         }
35     }
36 }
37 return z;
38 }
```

Time Analysis

We observe that we search for all the occurrences of pattern in the string in **linear traversal** of the string. Moreover once the right boundary and i^{th} pointer is increased, it's never reduced. So the **running time complexity of this algorithm is $O(n)$.**

END



Bitap Algorithm

Approximate string matching algorithm

Brief Overview

- It is an **approximate string matching** algorithm which compares two strings and finds out the **Levenshtein distance or Edit Distance** between them to find out the degree of dissimilarity between two strings.
- It can be broadly defined as *the minimum number of **edits (transition, addition, deletions)** in one string to make it equally as the other* using a very common programming paradigm generally known as **Dynamic Programming in $O(n^2)$** .
- There is some cost attached to each edit.



Applications

- Automatic spelling correction and replacing it from the correct word which has the lowest **Levenshtein distance** thus automating a lot of efforts which otherwise has to be done manually.
- Comparing the similarity in two DNA sequences which are nothing but a long string of A, T, G and C characters.
- It has applications in the field of bioinformatics and natural language processing.




Conversion of string A to B

Let us take an example to convert a string to other :

Let string **A** : “pqqrst” and **string B**: “qqttps”.

considering $C_A=1$, $C_T=1$ and $C_D=1$.

Let us define a function **F(String *initial*, String *final*)** which gives the **Edit Distance to convert *initial* to *final***. We will do any edit on the **last character** of the initial string be it addition, deletion or transition. Let us now for simplicity take the attached **costs as unity** for each edit and obtain a **recursive solution** for our problem.



$F(A, B) = F(\text{"pqqrst"}, \text{"qqttps"}) = \min (F(\text{"pqqrst\$"}, B) + C_A,$

$F(\text{"pqqr_"}, B) + C_D,$

$F(\text{"pqqrss"}, B) + C_T)$

Note : Here $F(\text{"pqqrss"}, B) + C_T = F(\text{"pqqrs"}, \text{"qqttp"}) + C_T$ as the last 's' is matched in both the strings so that is not needed to be changed.

_ indicates a blank string or *null* character.

When last characters are equal in both strings then the cost of transition is 0 because no transition is needed.



Now here we see a lot of duplication. The question is **can we reduce it down to a more compact form which can be easily coded?** Let's see...

Let's **modify F** to store only the required index of the string which indicates a point in string up to which the string has to be considered. So our new F looks something like this in a generalised form:

$$F(i, j) = \min (F(i + 1, j) + C_A,$$

$$F(i - 1, j) + C_D,$$

$$F(i - 1, j - 1) + C_T) .$$

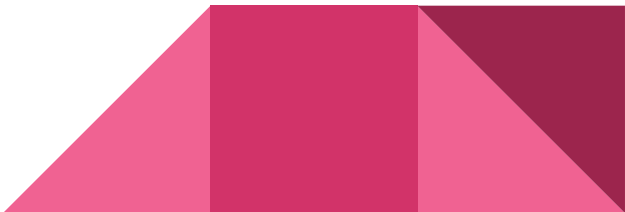
So just remember that the indices are signifying up to what point we are considering the string.



But here...do we see a problem??

We indeed can have a knowledge of the last states of a string, but having knowledge of the future states seems rather impractical, i.e. $F(\underline{i+1}, j)$.

We **don't know what that addition of character will lead to**. And anyway we'll have to delete that character to match the string length. We **need to convert this into an operation which we can see in the past**. So $F(i + 1, j)$ can be replaced by $F(i, j - 1)$ because **adding a character in A is equivalent to deleting a character in B**.

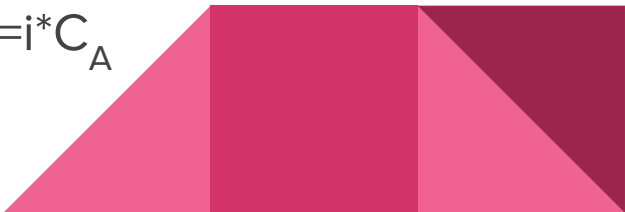


Final Recursive Solution...

So our **final recursive equation** which will lead us to solution becomes :

$$F(i, j) = \min \left(\begin{aligned} &F(i, j-1) + C_A, \\ &F(i-1, j) + C_D, \\ &F(i-1, j-1) + C_T \end{aligned} \right)$$

With base case $F(i, 0) = i * C_D$ and $F(0, i) = i * C_A$





Dynamic Programming Solution (Bottom to Top)

Now applying **Dynamic Programming** where the function F can have its elements stored in a 2-D array as:

$$F[i][j] = \min \left(F[i][j-1] + C_A, \right. \\ \left. F[i-1][j] + C_D, \right. \\ \left. F[i-1][j-1] + C_T \right)$$



Trace in a Grid (Example)

		i 						
j 		0	p	q	q	r	s	t
	0	0	1	2	3	4	5	6
	q	1	$\min(0+1, 1+1)=1$	1	2	3	4	5
	q	2	2	1	1	2	3	4
	t	3	3	2	2	2	3	3
	t	4	4	3	3	3	3	3
	p	5	4	4	4	4	4	4
	s	6	5	5	5	5	4	5

What Actually Happens..

So here is what actually happens to our string:

Moving right actually deletes a character from the first string

Moving down actually deletes a character from the second string

Diagonal move transitions character from one to another.



$[pqqrst, qqttps] \rightarrow [qqrst, qqttps]$ (Cost=1)

$[qqrst, qqttps] \rightarrow [qqrst, qqttps]$ (Cost=0)

$[qqrst, qqttps] \rightarrow [qqrst, qqttps]$ (Cost=0)

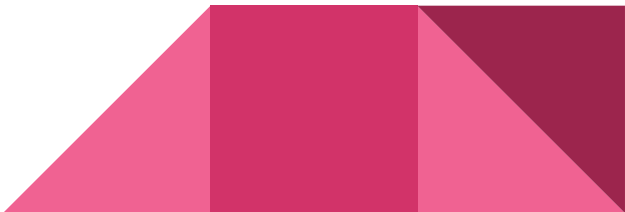
$[qqrst, qqttps] \rightarrow [qqtst, qqttps]$ (Cost=1)

$[qqtst, qqttps] \rightarrow [qqtst, qqtps]$ (Cost=1)

$[qqtst, qqtps] \rightarrow [qqtst, qqts]$ (Cost=1)

$[qqtst, qqts] \rightarrow [qqts, qqts]$ (Cost=1)

Thus the total Cost = $1+0+0+1+1+1+1=5$



C++ Code Implementation

```
1  int EditDistance(string s,string t)
2  {
3      //Need to find edit distance between s and t strings
4      int n=s.length();
5      int m=t.length();
6      int dp[n+1][m+1];
7      for(int i=0;i<n+1;i++)
8      {
9          for(int j=0;j<m+1;j++)
10         {
11             if(i==0)
12                 dp[i][j]=j;
13             else if(j==0)
14                 dp[i][j]=i;
15             else if(s[i-1]==t[j-1])
16                 dp[i][j]=dp[i-1][j-1];
17             else
18                 dp[i][j]=1+ min(dp[i][j-1],min(dp[i-1][j],dp[i-1][j-1]));
19         }
20     }
21     return dp[n][m];
22 }
```


Time Analysis

We observe that we have a recursive function which fills the grid of n^2 cells and each cell is filled just once so the time complexity of calculating edit distance or **Levenshtein distance** is **$O(n^2)$** .



Dictionary Implementation Using Trie

Brief Overview

We all have used a dictionary at least once in our life. But do we know how a computer lookup for the meaning of a word and that too very instantly?

Deep inside it uses a data structure known as **Trie**.

We will try to implement the same in this project. Using trie we can retrieve meaning in **$O(|s|)$ time complexity**.

It kind of uses a programming paradigm known as **dynamic programming**.



Structure of Trie Node

Each node will contain the following data:

- A bool which is set when a word ends on that node
- A string which carries the meaning of the word ending on that node, if the bool value is set.
- A map which maps characters (if required) from that node to others Trie Nodes as and when required.



Inserting a word in Dictionary

- To insert a string in the Dictionary we'll start from the root node and traverse through the trie following the map for each character.
- If we do not have a character mapped in the trie for the current character then we will insert the current character in the map which will point to a new Trie Node.



Finding meaning of a Word

- To find the meaning of a word we will start from root and follow the map each time for the current character.
- If we reach a null node and current pointer is not pointing to the end of the word then the **word do not exist** in the dictionary.
- If we do not end up on a null node but the bool is not set, still the **word do not exist** in the dictionary.
- If we do not end up on a null node and bool is also set then the **meaning of the word is stored in the string contained by that node.**



C++ code implementation

Trie Structure

```
struct Trie{
    bool isEndOfWord;
    unordered_map<char,Trie*> map;
    string meaning;
};

Trie* getNewTrieNode()
{
    Trie* node=new Trie;
    node->isEndOfWord=false;
    return node;
}
```

Insert Function

```
void insert(Trie*&root, const string& str, const string&meaning)
{
    if(root==NULL)
        root=getNewTrieNode();
    Trie*temp=root;
    for(int i=0;i<str.length();i++)
    {
        char x=str[i];
        if(temp->map.find()==temp->map.end())
            temp->map[x]=getNewTrieNode();
        temp=temp->map[x];
    }
    temp->isEndOfWord=true;
    temp->meaning=meaning;
}
```


Function to get meaning of a word

```
string getMeaning(Trie*root,const string&word)
{
    if(root==NULL)
    {return "";}

    Trie*temp=root;
    int n=word.length();
    for(int i=0;i<n;i++)
    {
        temp=temp->map[word[i]];
        if(temp==NULL)
            return "";
    }
    if(temp->isEndOfWord)
        return temp->meaning;
    return "";
}
```

Time Analysis

This algorithm has a worst case time complexity of $O(\sum |s_i|)$

END



References:

- <https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf>
 - <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8703383>
 - <https://core.ac.uk/download/pdf/26834073.pdf>
 - Wikipedia
 - Geeks for Geeks
- 