Project Report

On

Pattern Matching Algorithms

Submitted by

Namani Sreeharsh (180001032)

Ruchir Mehta (180001044)

Computer Science and Engineering

2nd year

Under the Guidance of

Dr. Kapil Ahuja

Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2020

CONTENT

ABSTRACT

We have chosen this theme to ensure us of the correctness of these algorithms and to learn deep mathematics behind these algorithms. We are desirous of learning how the text searching in text editors work, how are spell errors checked, etc. We will be analyzing the time and space complexities of the algorithms used in these practical cases and also implement them for our own satisfaction.

Manacher's Algorithm

(Longest Palindromic Substring)

Brief Overview

Manacher's algorithm finds the longest palindromic substring in a string using one of the famous and most used programming paradigm **Dynamic Programming** in the **time complexity of O(n)**. Palindromes are used at quite many places in real life implementing **compression algorithms**. There is research 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 the 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 example: let T: “#$a$b$a$b$a$b$a$@”

It has P: 00103050705030100

This is the naive approach which takes **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) → O(n).**

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

Procedure

Firstly let us realize that each palindrome has a **palindrome center** and a palindrome is always 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 the middle of the two characters. **In order to avoid such a situation** and to have a palindrome center such that all the palindromes in the string are explored, we need to **modify the string** and insert special characters like **#,$, and @.**

Hence, for example, any string “ababa” will be modified to **T: “#$a$b$a$b$a$@”**

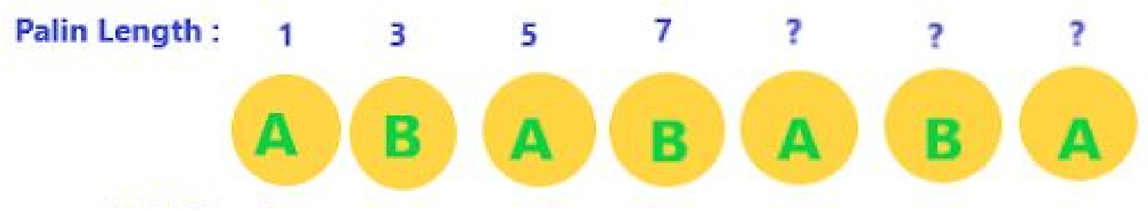
Terms and Definitions

Center (C): It is the center of the palindromic string which has been explored the right-most 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 a distance equal to the currently processing character from Center C to the left of C.

P array: Array storing the length of the palindrome with “i” as the center and having length P[i].



Since a palindrome is **symmetric about its center and within the 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 length of the palindrome around each center **before the center C**, we are not needed to explicitly calculate the palindrome length of characters **in the range of the farthest window**.

So let's maintain the **pointers:**

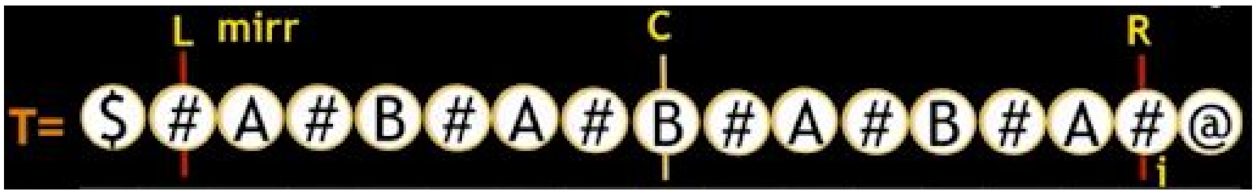
C: current center

r: right-most point

i: the current element to be explored

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:



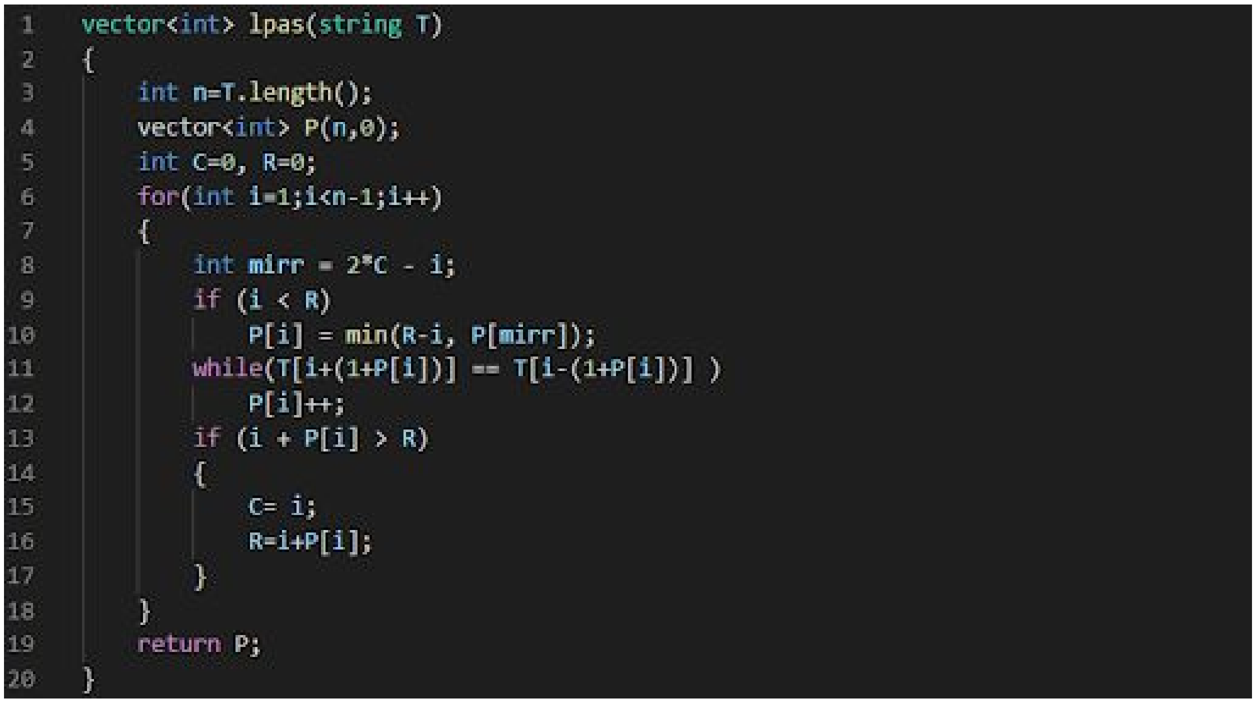
Algorithm

If the **answer** of current character as mirror length **exceeds the length of current** consideration of palindromic string then we'll **limit** our answer for that character to the **right boundary** of the currently considered palindromic string. If the answer of the current character as mirror length is within limit then we consider that as the current answer. Now we **expand** the palindromic length for the current character in **both directions** from the current palindromic length. Then we update the mirror position to the position of the current character if the palindromic length of current character exceeds the right boundary of the current palindrome into consideration.

Simulation

The **simulation** of the algorithm of constructing **P-Array** is hosted here.

Code Implementation



```
vector<int> lpas(string T)
{
    int n=T.length();
    vector<int> P(n,0);
    int C=0, R=0;
    for(int i=1;i<n-1;i++)
    {
        int mirr = 2*C - i;
        if (i < R)
            P[i] = min(R-i, P[mirr]);
        while(T[i+(1+P[i])] == T[i-(1+P[i])] )
            P[i]++;
        if (i + P[i] > R)
        {
            C= i;
            R=i+P[i];
        }
    }
    return P;
}
```

Time Analysis

We observe that we search for the longest palindrome in a **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).**

Bitap Algorithm

(**Approximate** string matching algorithm)

Brief Overview

It is an **approximate string matching** algorithm that 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 usually 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 that otherwise have 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"** consider 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("pqqrst" , "qqttps") = min (F("pqqrst$" , B) + CA,

F("pqqrs_" , B) + CD,

F("pqqrss" , B) + CT)

Note: Here F("pqqrss", B) + CT = F("pqqrs" , "qqttp") + CT 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 the 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. We **can reduce it down to a more compact form that can be easily coded**. 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 generalized form:

$$
\begin{aligned}
\mathbf{F(i\,,j) = min\,(\,F(i+1\,,j\,) + CA,} \\
\mathbf{F(i-1\,,j\,) + CD,} \\
\mathbf{F(i-1, j-1) + CT\,)\,.}
\end{aligned}
$$

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

Catch

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(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:

$$
\begin{aligned}
\mathbf{F(i\,,j) = min\,(\,F(i\,,j-1\,) + CA,} \\
\mathbf{F(i-1\,,j\,) + CD,} \\
\mathbf{F(i-1, j-1) + CT\,)}
\end{aligned}
$$

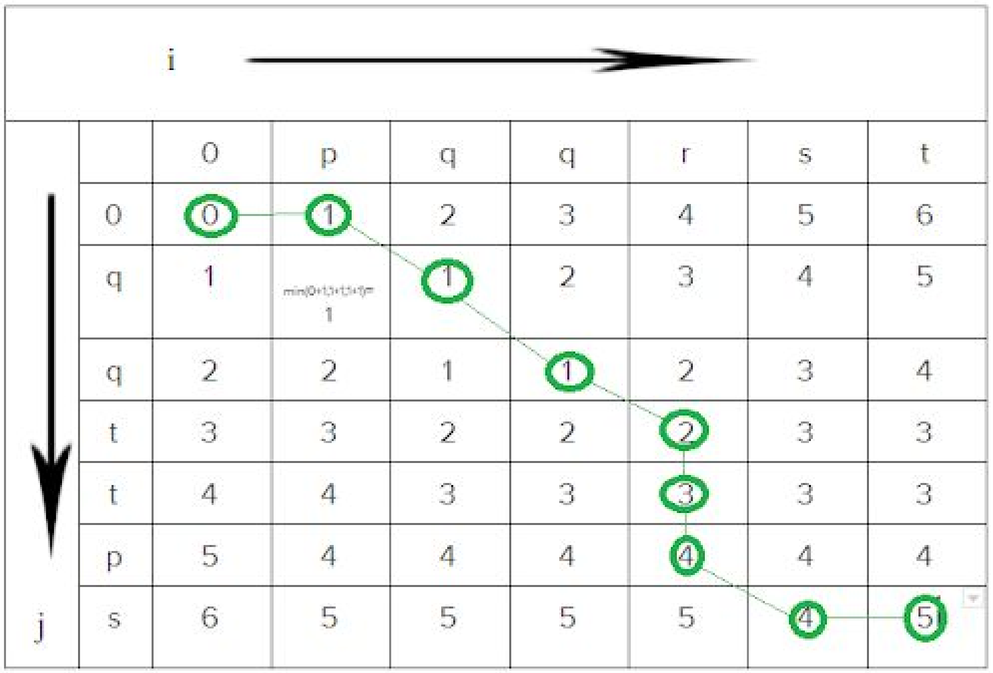
With base case $F(i,0)=i*C_D$ and $F(0,i)=i*C_A$

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:

$$
\begin{aligned}
\mathbf{F[i][j] = min\,(\,F[\,i\,][\,j-1\,] + CA,} \\
\mathbf{F[i-1\,][\,j\,] + CD,} \\
\mathbf{F[i-1][j-1] + CT\,)}
\end{aligned}
$$

Trace in a Grid (Example)



i

	0	p	q	q	r	s	t
0	0	1	2	3	4	5	6
q	1	min(0+1,1+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

j

Procedure followed

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]->[qqrst,qqttps] (Cost=1)

[qqrst,qqttps]->[qqrst,qqttps] (Cost=0)

[qqrst,qqttps]->[qqrst,qqttps] (Cost=0)

[qqrst,qqttps]->[qqtst,qqttps] (Cost=1)

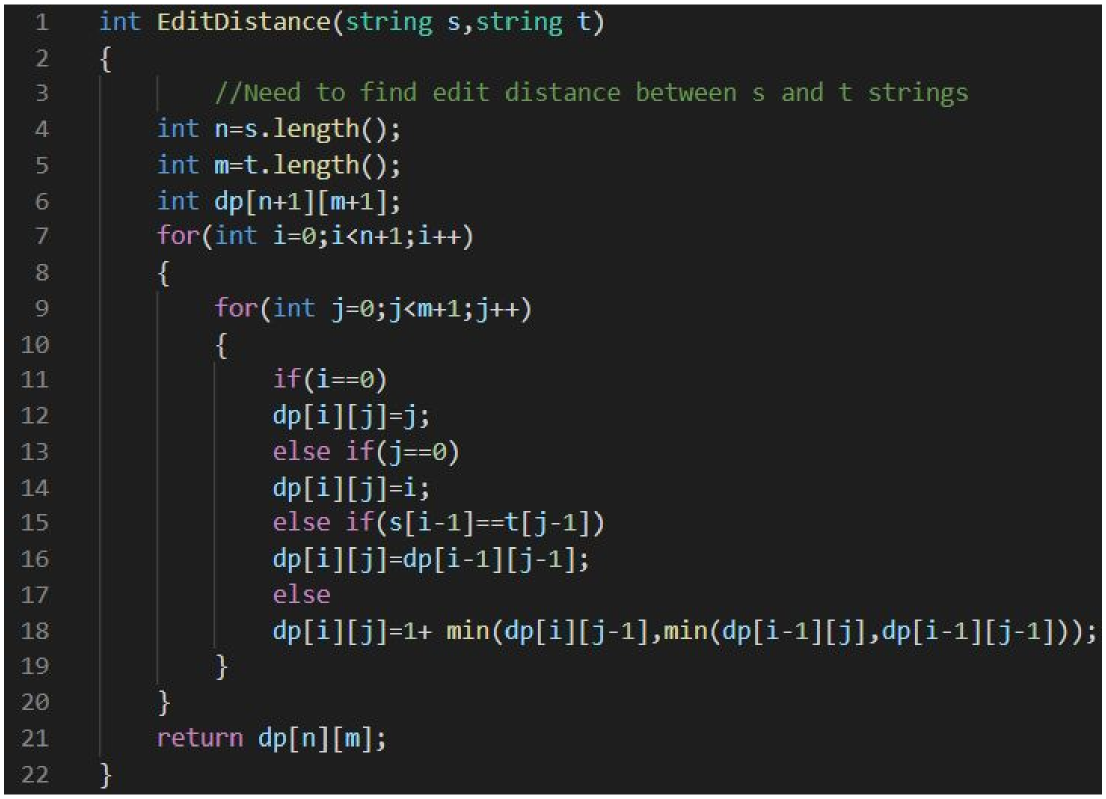
[qqtst,qqttps]->[qqtst,qqtps] (Cost=1)

[qqtst,qqtps]->[qqtst,qqts] (Cost=1)

[qqtst,qqts]->[qqts,qqts] (Cost=1)

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

C++ Code Implementation



```cpp
int EditDistance(string s,string t)
{
        //Need to find edit distance between s and t strings
    int n=s.length();
    int m=t.length();
    int dp[n+1][m+1];
    for(int i=0;i<n+1;i++)
    {
        for(int j=0;j<m+1;j++)
        {
            if(i==0)
            dp[i][j]=j;
            else if(j==0)
            dp[i][j]=i;
            else if(s[i-1]==t[j-1])
            dp[i][j]=dp[i-1][j-1];
            else
            dp[i][j]=1+ min(dp[i][j-1],min(dp[i-1][j],dp[i-1][j-1]));
        }
    }
    return dp[n][m];
}
```

Time Analysis

We observe that we have a recursive function that 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)**.

Rabin-Karp String Matching Algorithm

(**Pattern** matching algorithm)

Introduction

The Rabin Karp algorithm is a string matching algorithm, which is based on hashing the pattern and text in pieces of size equal to that of the pattern. The hash value of the pattern is the first step for filtering the chances of the text window being equal to the pattern. So, here we need to choose the best hash function which involves minimal collision.

Preliminary demystification of the algorithm

Terms used and their definitions

Window: Any substring of text tape whose size is equal to the size of the pattern.

Concept elicitation

The idea of hashing seems novel, provided the most efficient hash function is used. If the computation of the hash function of every window of text takes linear time then, roughly, the computational complexity of time would be of the order of the product of sizes of text and pattern, which again is rather inefficient. So, we need to come up with such a hashing scheme whose computation for every window is of constant order instead of linear order. Or in other words, the hash of a window should be extracted from the previous window in constant order time. This kind of hashing scheme is called roll hashing. As the word suggests, the hash value keeps rolling throughout the text. In this scheme, we need linear time just to find the hash of the first window. From this window, like a queue, we can remove the contribution of the leaving term or the first term of the window and the contribution of the upcoming new character. So, this step will be of constant order if the contributions are united in the hash function by addition. And the removal of their contribution means, subtracting the term containing the leaving term and adding a new term having the characteristics of the new character. This gives the hash value of the second window in constant time instead of O(m) time, again.

Intricate analysis

Basically, once we decide the best hash function, we are done. If we represent the characters as some numbers and if we write the expanded form of the obtained number with some base, then, we will get minimum collisions. If the hash function is mere addition of all number codes of characters, then, there are more chances of collision as a set of entirely different numbers i.e, with no element between the two sets common, also gets the same hash. So if the hash function is in the form of a polynomial of a single variable with some base, then, it would be an extremely better hash function than the mere sum, as collisions are very less for this hash function. But for patterns of vivid characters or large length, there might be memory overflow as the domain of the hash function is large. So, to minimize collision, the leading term of the polynomial base raised to the power of (number of distinct characters in pattern -1) will also be high. Though in theory, it isn't a problem, but in pragmatic aspects, it is very difficult to hold by the computer, as computers, in general, can store only integers till the predecessor of 2 raised to the power of 63. So for preventing overflow, we always take the modulus of outcomes at each step of computation by dividing the outcomes of each step with a large number and obtaining the hash. So, this increases the chances of frequent collisions thus resulting in an increase of probability of tending towards the worst case of the algorithm, in terms of time, whose computational complexity is of the order of the product of text size and pattern size. So to tackle this problem, we have 2 cases:

Case 1

Using strings. But while using strings, we need to define all operations needed to be performed on the numbers so that the requisite snippets can be simulated by preparing functions for them. The operations to be simulated are addition, power, multiplication, comparator which simply says whether the two numbers being handled as strings by the compiler are merely equal or unequal and nothing else. All the operations can be done in linear time except multiplication. As in multiplications, we need to multiply numeric codes and base raised to the powers of integers from 0 to the predecessor of the number of distinct characters in the pattern string. In this multiplication, the base raised to the powers has more number of digits than numeric codes of the characters of the text. So, the computational complexity of this multiplication, if Karutsuba's divide and conquer algorithm is used, is of the order of m raised to the power of log2 (3) thus increasing the complexity to n*m^(log$_2$(3)) as in roll hashing, we always multiply every window's first numeric code with base raised to the power of predecessor of the number of distinct characters of pattern and subtract from the current window's hash and multiply with base and add numeric code of the incoming character to it to get the hash value of next window. This step is done n-m times, so is the complexity.

Case 2

Degrading the hash function by taking the modulus of it when divided by a big number at every step and using standard primitive data types of integers and applying normal arithmetic operations defined in the compiler. This is better than the above approach as it doesn't degrade the computational complexity to the extent of being worse than the Brute Force. As we use standard operators freely, the complexity of finding hash of every window apart from the first one is of constant order as all the operations performed like multiplying every window's first numeric code with base raised to the power of predecessor of the number of distinct characters of pattern and subtracting from the current window's hash and multiplying with base and adding numeric code of the incoming character to it to get the hash value of next window are of constant order.

Time complexity and space complexity:

Time complexity is of order O(m+n) in average and best case but O(m*n) in the worst case. Because if each and every window has got the pattern then, the algorithm performs character by character search for n-m+1 windows, no matter how efficient the hash function is.

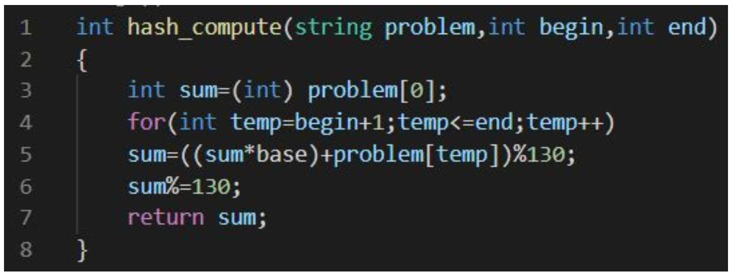
So this algorithm definitely scores over naive as in most of the cases, where the pattern is sparsely present in the text, it takes O(m+n) on average to complete the algorithm.So advantageous for the situations in which the pattern occurs rarely in the text.

Space complexity is O(1) as this algorithm uses no data structure to manipulate the data.

C++ code snippets

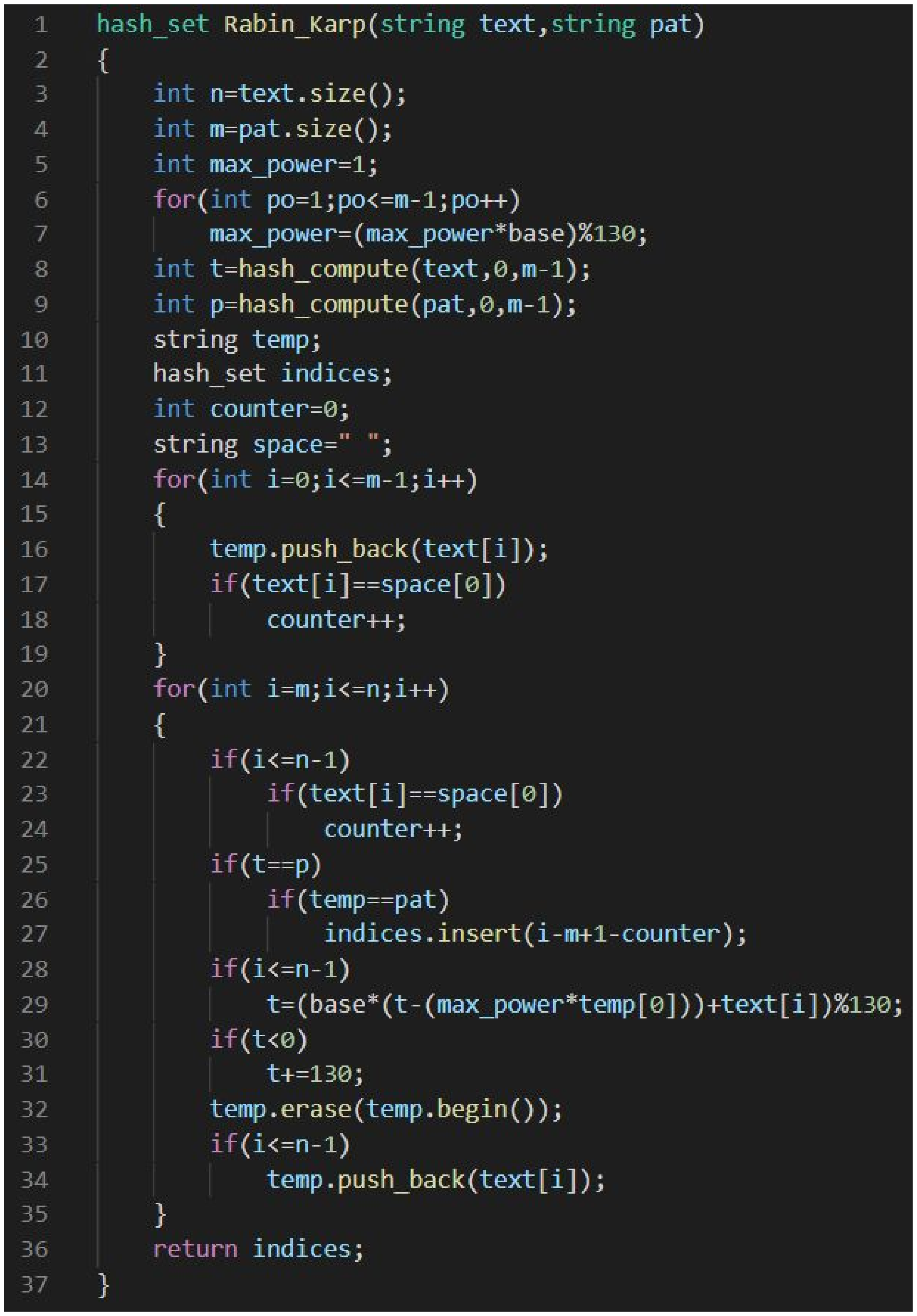
Note: 130 is overflow controlling number so, in any step of computation, we won't get any number greater than or equal to 130

Computing the hashes of the first window of the text and the pattern



```
int hash_compute(string problem,int begin,int end)
{
    int sum=(int) problem[0];
    for(int temp=begin+1;temp<=end;temp++)
    sum=((sum*base)+problem[temp])%130;
    sum%=130;
    return sum;
}
```

The prime component of the algorithm



```
hash_set Rabin_Karp(string text,string pat)
{
    int n=text.size();
    int m=pat.size();
    int max_power=1;
    for(int po=1;po<=m-1;po++)
        max_power=(max_power*base)%130;
    int t=hash_compute(text,0,m-1);
    int p=hash_compute(pat,0,m-1);
    string temp;
    hash_set indices;
    int counter=0;
    string space=" ";
    for(int i=0;i<=m-1;i++)
    {
        temp.push_back(text[i]);
        if(text[i]==space[0])
            counter++;
    }
    for(int i=m;i<=n;i++)
    {
        if(i<=n-1)
            if(text[i]==space[0])
                counter++;
        if(t==p)
            if(temp==pat)
                indices.insert(i-m+1-counter);
        if(i<=n-1)
            t=(base*(t-(max_power*temp[0]))+text[i])%130;
        if(t<0)
            t+=130;
        temp.erase(temp.begin());
        if(i<=n-1)
            temp.push_back(text[i]);
    }
    return indices;
}
```

KMP Algorithm

(**Pattern** matching algorithm)

Introduction

Knuth-Morris_Pratt algorithm is an effective replacement of naive algorithm as it makes use of a property of proper prefixes and suffixes which brings down the complexity of the algorithm from O(m*n) to O(m+n) where m is the length of the pattern to be explored in the whole text and n is the length of the whole text itself. It is also called the K.M.P. algorithm in short. One of its obvious applications in the implementation of Ctrl-F functionality in computers for finding all the occurrences of a particular substring in the whole text (in pdf files,.docx files, .txt files, etc.,). Its ease of implementation also promotes its utility in various fields demanding huge application of string matching.

Preliminary demystification of the algorithm

Terms used and their definitions

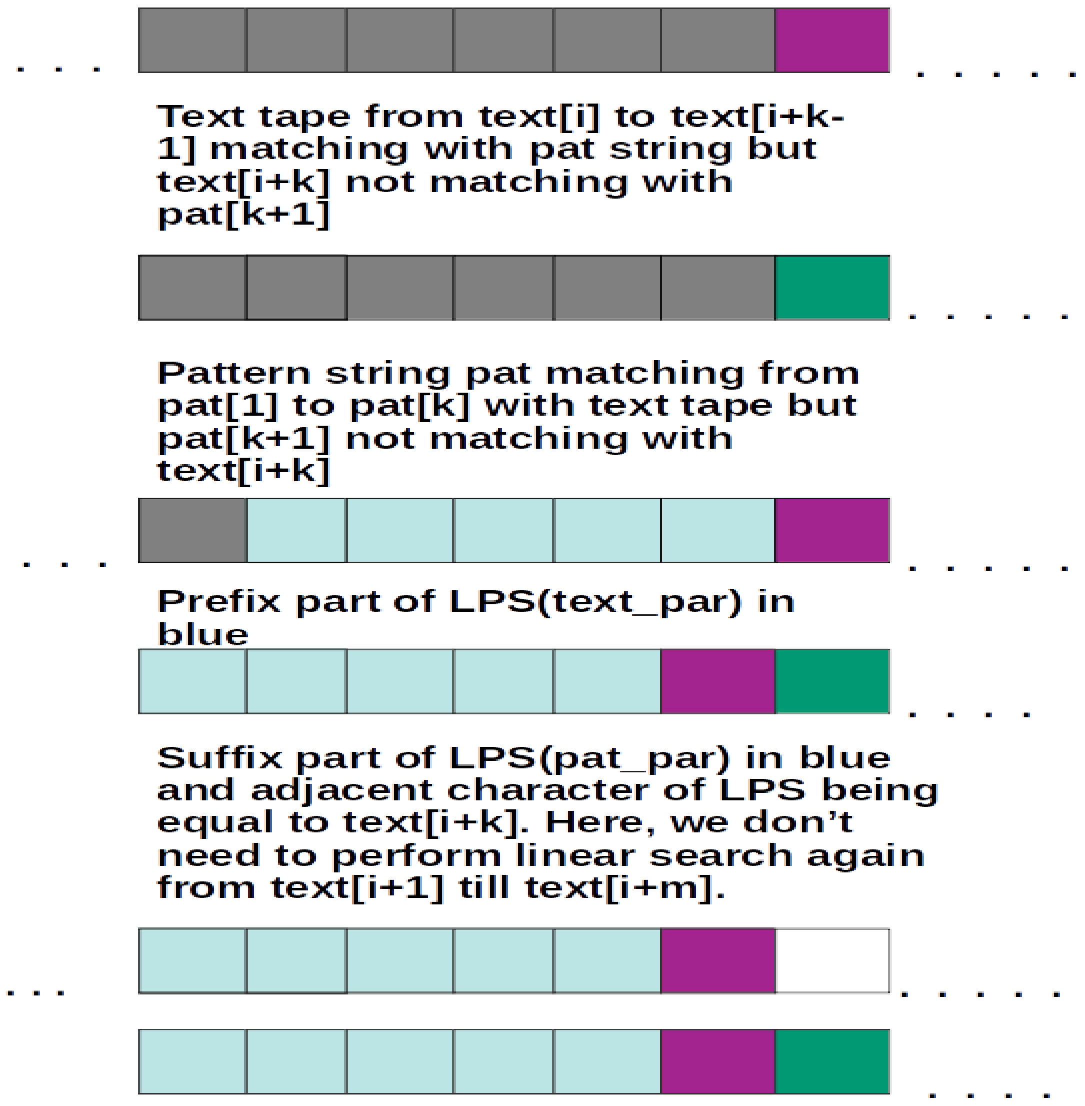
Proper prefixes of a string: All prefixes of the given string excluding the whole string itself.

Proper suffixes of a string: All suffixes of the given string excluding the whole string itself.

Longest proper prefix and suffix: The longest possible string amongst all possible proper suffixes and prefixes which is both proper prefix and proper suffixes simultaneously. It will be expressed as LPS in short in upcoming elicitations. It can be expressed as a function that returns LPS if present, else returns a value indicating exception.

Prefix part of LPS(string): LPS of a string can be realized as either prefix or suffix as per the definition. So if we need to realize LPS as a prefix of the string while visualizing its presence in the string whose LPS is being computed, we state its realization as a prefix, by mentioning it as a prefix part of LPS(string) in further discussions.

Suffix part of LPS(string): LPS of a string can be realized as either prefix or suffix as per the definition. So if we need to realize LPS as a suffix of the string while visualizing its presence in the string whose LPS is being computed, we state its realization as a suffix, by mentioning it as a suffix part of LPS(string) in further discussions.


Text tape from text[i] to text[i+k-1] matching with pat string but text[i+k] not matching with pat[k+1]
Pattern string pat matching from pat[1] to pat[k] with text tape but pat[k+1] not matching with text[i+k]
Prefix part of LPS(text_par) in blue
Suffix part of LPS(pat_par) in blue and adjacent character of LPS being equal to text[i+k]. Here, we don't need to perform linear search again from text[i+1] till text[i+m].

Concept elicitation

Consider a string "text" of length n in which we need to explore all patterns of another string "pat" of length m. If both of them match from text[i] to text[i+k-1] in text dataset and pat[1] to pat[k] where k is the length of the current match(k<m) and i is the point from which it started matching with the pattern string partially. Then if pat[k+1]=text[i+k] then we have no problem, we can move ahead by increasing k by 1 and solving the further independent case. But if pat[k+1]!=text[k] then in naive approach, we simply restart linear search from text[i+1] to text[i+m]. This is where K.M.P. scores over naive. We can make use of the information extracted from the strings which are equal till the point of inequality of end characters (pat[k+1] and text[i+k]). Let us denote partial strings as text_par (from text[i] to text[i+k-1]) and pat_par (from pat[1] to pat[k]). LPS(pat_par)= LPS(text_par) as pat_par = text_par . So, if we find the immediate adjacent character of the prefix part of LPS(pat_par) and compare it with text[i+k], then, we can continue the problem further as we found a chunk of a prefix of the pattern matching with the text again. From here, we again get an

independent case which needs to be solved by the same procedure and if there is no LPS (text_par) or LPS(pat_par) and last characters are unequal then, we need to move ahead in the text tape by 1 and again start exploring for the pattern from scratch. Once we find the whole pattern needed in the text, we need to again move the pattern pointer to index of the last element of LPS(pat)+1 for knowing the further occurrences of the pattern and continuing to apply the same algorithm.

Intricate analysis

Processes to be dealt with are:

1. Preprocessing the pattern.
2. Processing the text string.

Preprocessing the pattern

As we can observe, an efficient sub-algorithm for finding LPS of all prefixes of the pattern is very crucial for the algorithm because the entire algorithm depends on this step. So, we need to precompute all 1+index of LPS of all prefixes and store them in a table before actually processing the text itself and the table obtained is known as Pi table. It should be a hash table for looking up the index of LPS(pat_par)+1 of all prefixes whenever needed as complexities of insertion and value extraction operations are of constant order, provided the hash table deals with collision perfectly by using the best possible hash function.

Following 0 indexing policy of strings from here.

In 0 indexing policy of arrays and strings, the number of elements from the first element to that element is equal to the index of the element immediately after the range. So, from here the index of LPS(pat_par)+1 is nothing but the size of LPS as LPS is always considered as a prefix in pat_par which starts from the 1st element of the pattern.

The logic for this step is :

For prefix of size 1, no proper prefix and suffix is possible so for that prefix, LPS is by default 0

For prefix of size 2 if the first element of pat is equal to the second element, then the length of LPS is 1 as only 1 proper prefix and 1 proper suffix are possible else length of LPS is 0

For prefixes of size greater 2, we can know the size of LPS from the LPS information of prefix whose length is just 1 less than itself.

Consider prefix and suffix parts of LPS of the previous prefix. If LPS of the previous one does not exist i.e, 0 and if 1st character is not equal to the last character of the prefix then, it's LPS is also non-existent and its value is also 0. Else if one character just after the prefix and suffix parts are equal then we just need to simply increase the previous one by 1 as the resultant strings in both parts will form the longest proper prefix and suffix for the current prefix and their size is just 1 more than the previous one. Otherwise, we need to compare the character just after prefix part of LPS(prefix part of LPS(previous prefix)) and just after suffix part of LPS(suffix part of LPS(previous prefix)) i.e, the last character of the original prefix itself and continue till they are not equal and increase by 1 after getting equality case if equality case is not attained then, we need to follow 1st condition of this case.

The third case of the logic follows the tabulation approach or bottom-up approach of dynamic programming as the solution is built incrementally using the previous details. Though at first look, we may feel the presence of a backtracking approach, in reality, it isn't. For every prefix we try to be greedy i.e., we try to take the largest LPS on which we can build the current prefix by comparing the character just after the prefix part of LPS(previous prefix) and the last character of the current prefix for which we are computing the size of its LPS and once the equality condition is found, we stop digging into the LPS further and in the entire process, we do not move in the opposite direction of exploration thus backtracking being absent. Thus this step is an amalgamation of greedy and dynamic programming strategies.

The complexity of the construction of Pi table is **O(m)** as for every element of pattern, we calculate the size of LPS of the prefix ending at the character from the previous results.

Video hosted in this link.

Processing the text string

If pattern and text match to some extent and if the latest characters of pattern and the text do not match then, we need to compare the character just after the prefix part of LPS(pat_par) and the latest text character and proceed as discussed in concept elicitation section of preliminary demystification of the algorithm. As we get LPS of a prefix in the order of constant time from Pi table, we can extract all the needed patterns in the order of n i.e, the size of the whole text itself, which is the minimum time complexity for finding all the occurrences of a particular pattern.

For an incomplete pattern match case here is the hosted video.

For a complete pattern match case, here is the hosted video.

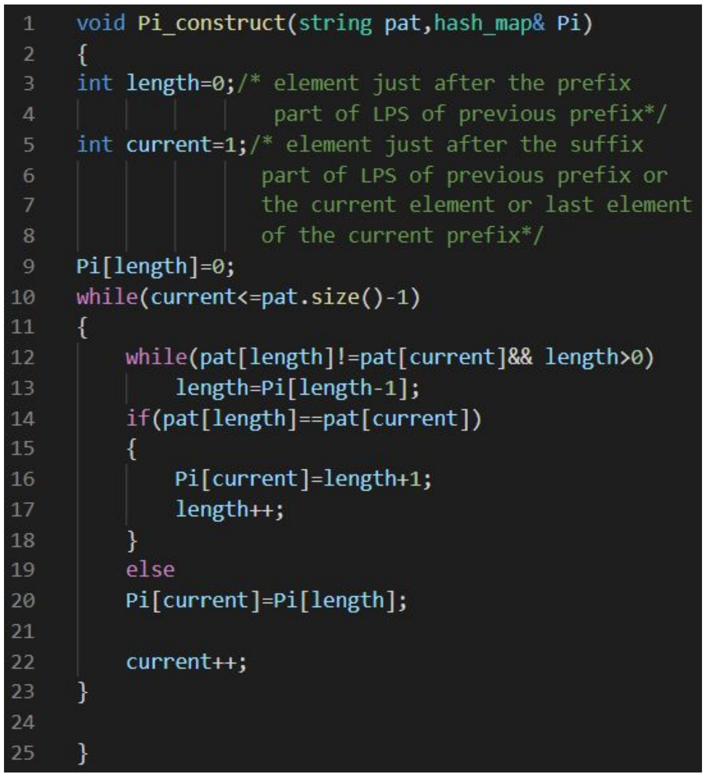
Time complexity and Space complexity:

Time complexity of the preprocessing phase is O(m) and the text processing phase is O(n) where m and n are sizes of pattern and text respectively. So time complexity of the entire algorithm is O(m+n).

Space complexity of the algorithm is O(m) as the Pi table for pattern is prepared in the form of a hash table in the algorithm.

C++ code snippets

Construction of Pi table

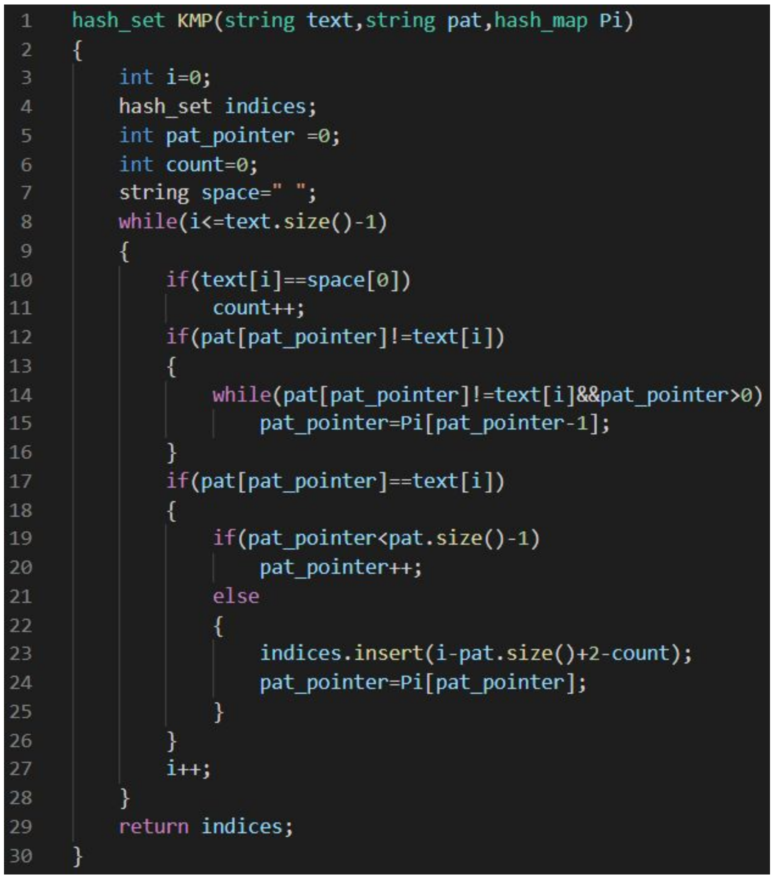


```cpp
void Pi_construct(string pat,hash_map& Pi)
{
int length=0;/* element just after the prefix
                part of LPS of previous prefix*/
int current=1;/* element just after the suffix
               part of LPS of previous prefix or
               the current element or last element
               of the current prefix*/
Pi[length]=0;
while(current<=pat.size()-1)
{
    while(pat[length]!=pat[current]&& length>0)
        length=Pi[length-1];
    if(pat[length]==pat[current])
    {
        Pi[current]=length+1;
        length++;
    }
    else
    Pi[current]=Pi[length];

    current++;
}

}
```

For Text Processing



```
hash_set KMP(string text,string pat,hash_map Pi)
{
    int i=0;
    hash_set indices;
    int pat_pointer =0;
    int count=0;
    string space=" ";
    while(i<=text.size()-1)
    {
        if(text[i]==space[0])
            count++;
        if(pat[pat_pointer]!=text[i])
        {
            while(pat[pat_pointer]!=text[i]&&pat_pointer>0)
                pat_pointer=Pi[pat_pointer-1];
        }
        if(pat[pat_pointer]==text[i])
        {
            if(pat_pointer<pat.size()-1)
                pat_pointer++;
            else
            {
                indices.insert(i-pat.size()+2-count);
                pat_pointer=Pi[pat_pointer];
            }
        }
        i++;
    }
    return indices;
}
```

Automation String Matching Algorithm

Introduction

The algorithm of string matching using finite automata is more efficient than the K.M.P. string matching algorithm once the Deterministic Finite Automata is prepared. An automata machine checks the state of the text tape while checking each character and once the final state in the automaton is reached, we can insert the index of the initial character in the text in the set and return the set. The algorithm has 2 parts in it. Constructing a D.F.A. and using the D.F.A. to find all the occurrences of the pattern in the text or text processing phase.

Preliminary demystification of the algorithm

Terms and definitions used in the text

Pi Table: The preprocessed table obtained at the end of the preprocessing phase of the pattern in the K.M.P. algorithm.

Traversal cases: Cases covering a few instances of inequality of the character of the pattern being pointed to and the latest text character, instructing the final position of the pattern pointer post inequality discovery i.e, though inequality is observed, on getting some letters, we can know where to move the pattern pointer without performing an additional comparison. If the obtained text character is unequal to pattern character and is not present in the cases mentioned below the pattern character then, it means that the pattern pointer must be moved to the first character of the pattern again. It is represented as a hashmap of characters and numbers where characters being the key and the numbers being the position of the pattern pointer if that case occurs. And the set of these hash maps for each character is represented as an array where i^{th} hash_map denotes the traversal cases of the ith element of the pattern. This is the transition graph of D.F.A.(Deterministic Finite Automata) as well and the graph is represented as an array of hash maps for every character of the pattern and

Case: A key-value pair, in any set of traversal cases containing the information regarding the final position of pattern pointer, as key, on getting its corresponding key character instead of the character present in the index, whose set of traversal cases is being processed.

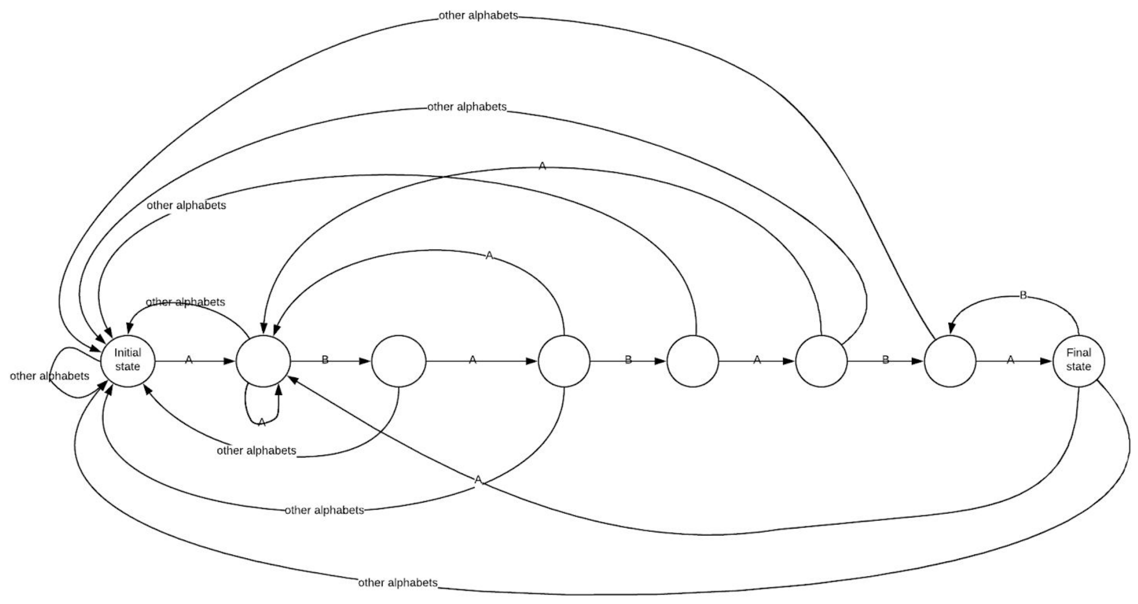
Concept elicitation

When a D.F.A. is to be prepared, we must ensure that every state has all possible transitions for all the alphabet. And on keen observation, we can say that in D.F.A., we move back from one state to the next state of a previous state on the basis of the substring which can be obtained from the initial state to that particular previous state. Once we find that substring, we move to the next state of that particular previous state, thus reducing the rechecking efforts. This is again determined from the Largest proper Prefix and Suffix(L.P.S.).The same concept is used in the K.M.P. algorithm. If the strings and text tape are matching to some extent, then, on the basis of the next character of L.P.S. of the partial equal part or prefix in the pattern being equal to the latest character of the text being pointed to, we move to the next state of the previous state. But, in this part, we have to backtrack in the pattern string thus increasing the number of steps in text processing part as, we have to move back the pattern pointer until the character at which it is pointing to, is equal to the latest character in the text tape, using the Pi table, with no movement of the text tape pointer. But with the additional work of noting down all the cases i.e, to which successor of the L.P.S. of the prefix of the pattern pointer should approach on obtaining certain text characters, which are not equal to the one being pointed to in the text instead of just writing where to go in the case of inequality irrespective of the unequal character found, we can prevent this case of non-deterministic pattern backtracking as well by making use of the L.P.S. information and successive characters of all L.P.S.s of all the prefixes before the current partial equal string or current prefix by drawing a different table from K.M.P.. By doing this, we can always decide the next state of traversal in the D.F.A. in all the circumstances i.e., in the cases of equality or inequality with the latest text character in only one step unlike K.M.P., thus acting deterministically at each step or state.

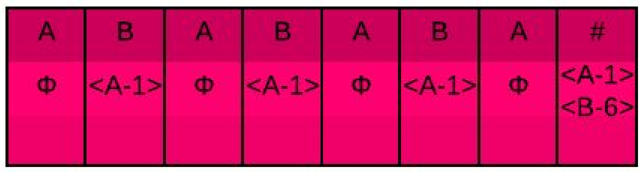
Intricate analysis

1. Preprocessing phase

In the preprocessing phase, after constructing Pi table, we need to know for every character of the pattern, what are the successors of the L.P.S. of the string from the first letter of the pattern to the character just before the current character and copy the traversal cases of that character and assign to that of the current character and remove the particular case from the traversal cases of the current character which is equal to the current character and add the case of the successor of the L.P.S. of the prefix just before the current character in the pattern, if it is not equal to the current character. This clearly can be done in O(m*number of distinct characters or the size of the alphabet set) by constant space tabulation dynamic programming approach like the one implemented in the construction in the Pi table(for finding the size of L.P.S. of the current prefix from the one that of the previous prefix). As any hash table can have the maximum size of that of the alphabet set, the copying complexity of that hashmap can never be greater than the size of the alphabet set. The final state of the D.F.A. is represented by one more additional character at the end of the string, by appending the character to the pattern, at the end, which is not at all used in the text and after reaching that state, we do the insertion operation of the starting index into the set that is returned by the algorithm and as the appending character does not appear in the text, we follow the traversal cases of that character for the character in the text tape, just after the latest text character being pointed to, and we mimic the behavior of a D.F.A. machine. Thus, the graph has m+1 number of states in it.



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.

Simple logic is Graph obtained is a guide book for the case of violations whereas the pattern string itself is a guide book for matching cases. If pattern pointer and text pointer point to the same character then we increment both pointers by 1 else, we follow the graph instructions. If the encountered violation is not a traversal case in the graph, then we simply move the pattern pointer to the first character of the pattern and continue the algorithm.

2. Text processing phase

Text processing phase takes exactly only O(n) for each text character, we use only one character comparison unlike K.M.P., which sometimes needs to backtrack in pattern taking more than 1 step thus resulting in more computational steps.

3. Time complexity analysis

The total computational complexity of the algorithm as a whole is O(m*(alphabet_size)+n) where m*(alphabet_size) is due to the construction of the graph of D.F.A. whereas n is due to the whole text processing and is O(m+n) as alphabet_size has an upper bound of 256 as the number of possible ASCII characters accepted is 256 and in practice, we don't use more than ASCII characters. So, alphabet_size would rather be a constant and it can be written as above.

4. Space complexity analysis

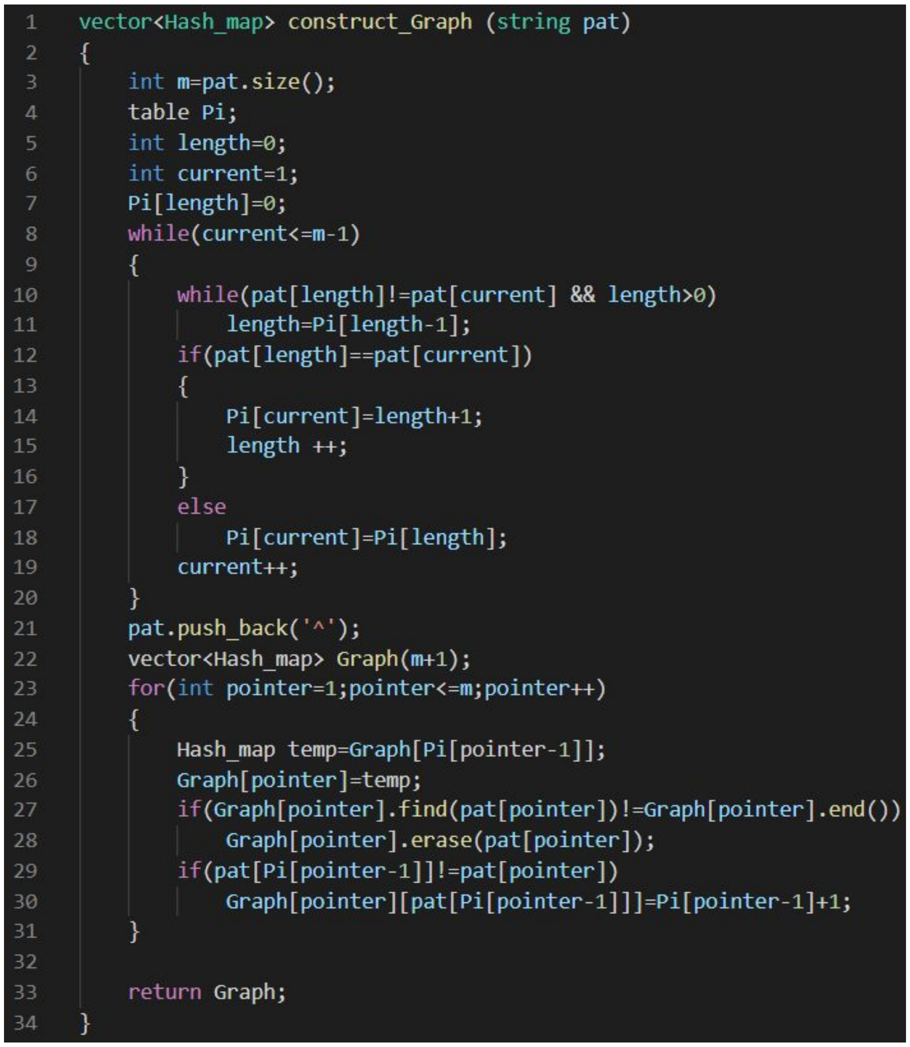
Structures stored while using the algorithm:

1. Pi table(a single hash table)
2. Graph for D.F.A. which needs to be processed in text processing phase (vector or array of hash tables)
3. Constant order space for storing variables like pointers in the text tape and pattern and prefix and suffix holders during the construction of Pi table
4. structure needs O(m) as it has indices of the pattern as key and all indices are present in the hash table.
5. It is basically an array of hash tables and each hash table has at most alphabet_size number of entries and there are m+1 such hash tables. So space of occupancy is O(m*(alphabet_size))=O(m) as alphabet_size is less than or equal to 256, in practice.
6. There are of O(1) as number of such constant space are not a function of n or m or alphabet_size

   So the space complexity of the algorithm is O(m).

C++ Code Snippets

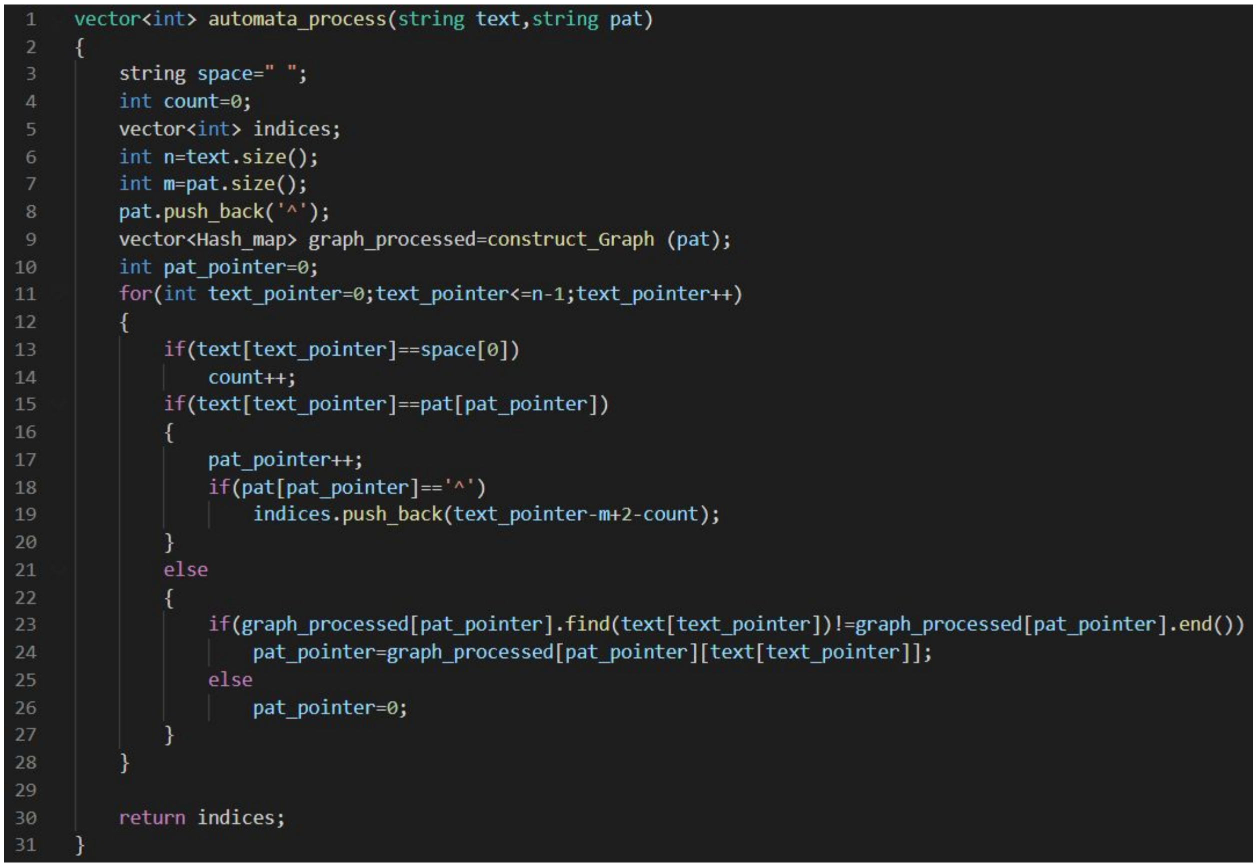
Constructing the D.F.A. Transition Graph



```cpp
vector<Hash_map> construct_Graph (string pat)
{
    int m=pat.size();
    table Pi;
    int length=0;
    int current=1;
    Pi[length]=0;
    while(current<=m-1)
    {
        while(pat[length]!=pat[current] && length>0)
            length=Pi[length-1];
        if(pat[length]==pat[current])
        {
            Pi[current]=length+1;
            length ++;
        }
        else
            Pi[current]=Pi[length];
        current++;
    }
    pat.push_back('^');
    vector<Hash_map> Graph(m+1);
    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;
}
```

For text processing



```
vector<int> automata_process(string text,string pat)
{
    string space=" ";
    int count=0;
    vector<int> indices;
    int n=text.size();
    int m=pat.size();
    pat.push_back('^');
    vector<Hash_map> graph_processed=construct_Graph (pat);
    int pat_pointer=0;
    for(int text_pointer=0;text_pointer<=n-1;text_pointer++)
    {
        if(text[text_pointer]==space[0])
            count++;
        if(text[text_pointer]==pat[pat_pointer])
        {
            pat_pointer++;
            if(pat[pat_pointer]=='^')
                indices.push_back(text_pointer-m+2-count);
        }
        else
        {
            if(graph_processed[pat_pointer].find(text[text_pointer])!=graph_processed[pat_pointer].end())
                pat_pointer=graph_processed[pat_pointer][text[text_pointer]];
            else
                pat_pointer=0;
        }
    }

    return indices;
}
```

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 the KMP algorithm also performs the same task, we want to focus on these 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 applications 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.

Simulation

The **simulation** of the algorithm of constructing the **Naive Pattern Matching Algorithm** is hosted **here.**

Analysis of naive approach

We realize that the **brute approach has $O(n^2)$** time complexity in the worst case but **we can reduce this to O(n)** time complexity using some preprocessing. This can be achieved using **Z-Algorithm** which creates the **Z-array** as a **preprocessing step**. Firstly let's 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 ith term i.e. **Z[i] represents the length of the longest substring starting from the ith index which is also the prefix of the string**.

For example: **"abacabfab"** has Z array as:

Z: **001020020**

Note: Z[0] is always equal to the length of the string but for calculation purposes, we take Z[0]=0.

Once we know how to calculate **Z-Array in O(n)**, we use it to **find all possible positions** where a **pattern matches in a string.**

From Z-Array -> 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 the length of the pattern**. This **ith index is the starting point** of the matched pattern in the string T. To get an index **in S,**

simply subtract (len(P)+1) from i.

Terms and Definitions

i: current pointer to the character who’s Z[i] is to be calculated

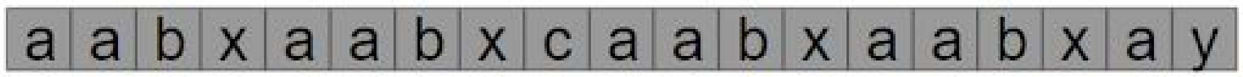
Z: Array where Z[i] represents the length of the longest string starting at ith element which is also the prefix of the string.

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….



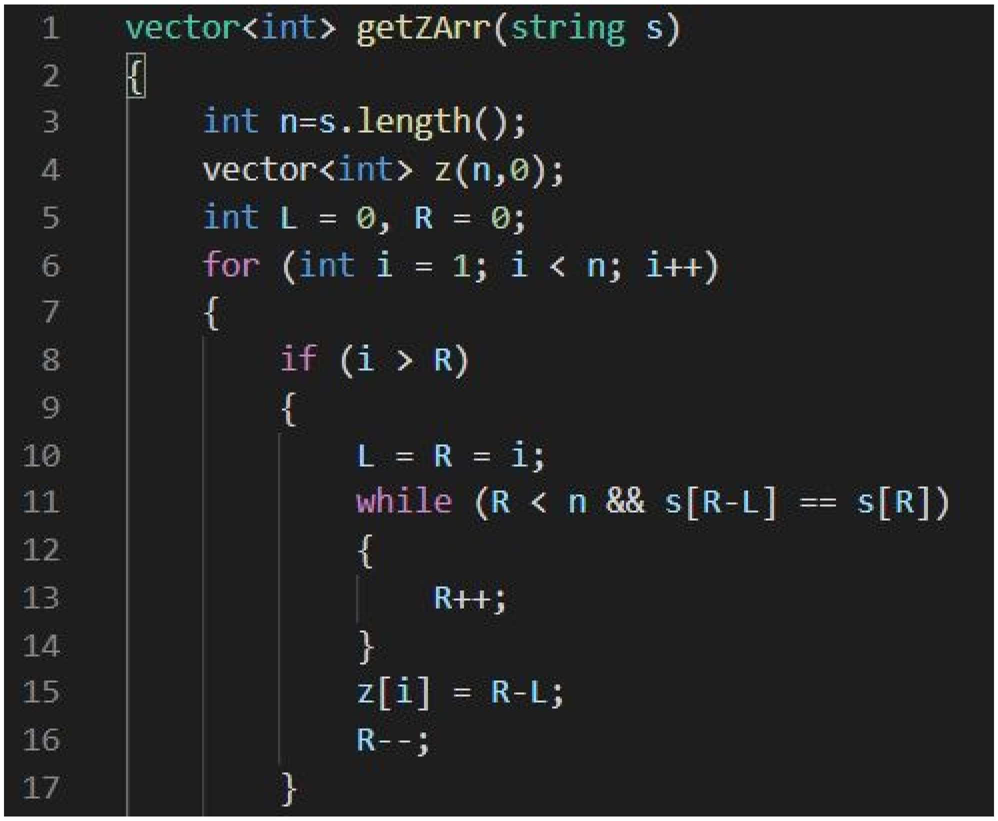
a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y

Here..in simulation.

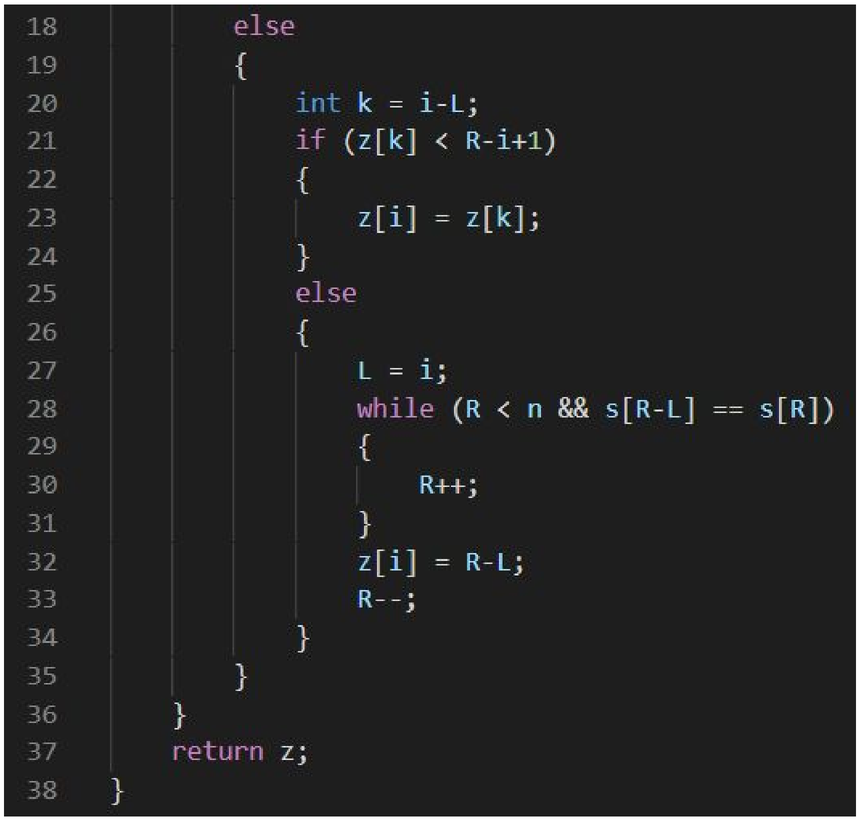
Simulation

The **simulation** of the algorithm for the formation of Z-Array is hosted **here**.

Code Implementation in C++



```cpp
vector<int> getZArr(string s)
{
    int n=s.length();
    vector<int> z(n,0);
    int L = 0, R = 0;
    for (int i = 1; i < n; i++)
    {
        if (i > R)
        {
            L = R = i;
            while (R < n && s[R-L] == s[R])
            {
                R++;
            }
            z[i] = R-L;
            R--;
        }
```



```cpp
        else
        {
            int k = i-L;
            if (z[k] < R-i+1)
            {
                z[i] = z[k];
            }
            else
            {
                L = i;
                while (R < n && s[R-L] == s[R])
                {
                    R++;
                }
                z[i] = R-L;
                R--;
            }
        }
    }
    return z;
}
```

Time Analysis

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

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

Inserting a word in Dictionary

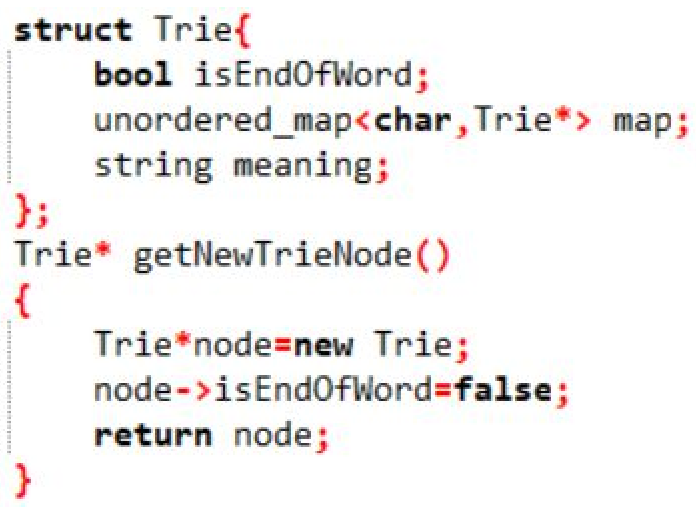
To insert a string in the Dictionary we'll start from the root node and traverse through the trie following the map for each character. If we do not have a character mapped in the trie for the current character then we will insert the current character in the map which will point to a new Trie Node.

Finding the meaning of a Word

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

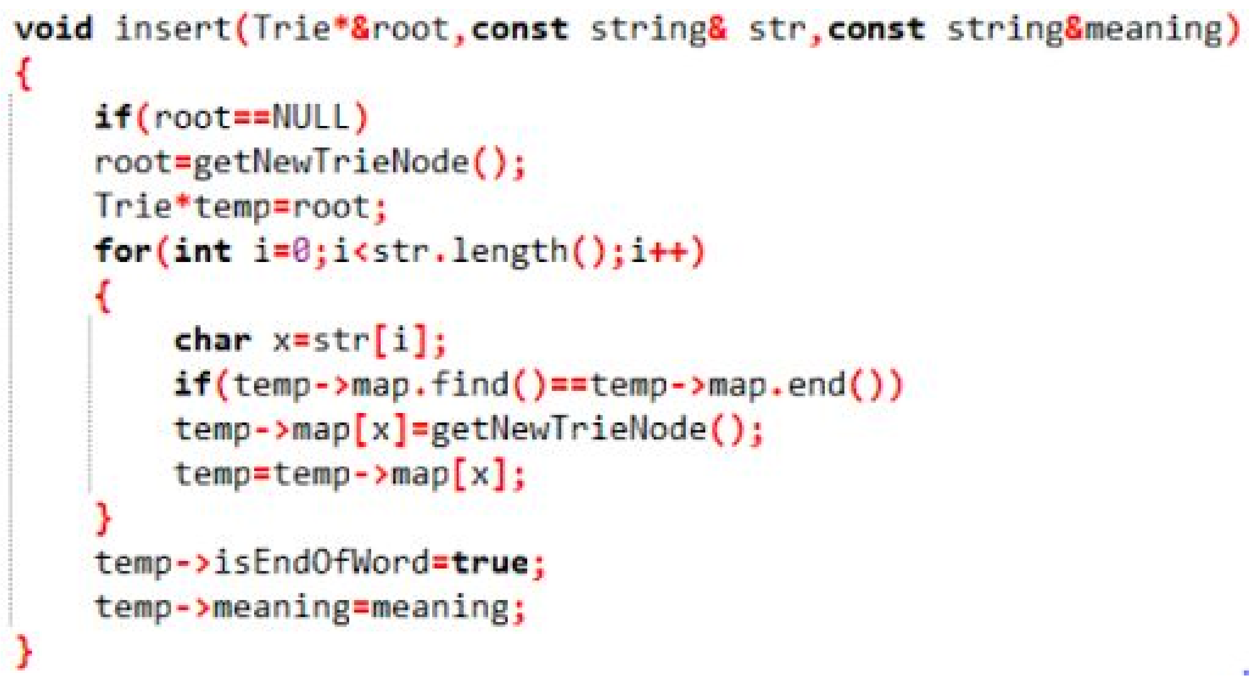
C++ code implementation

TrieStructure



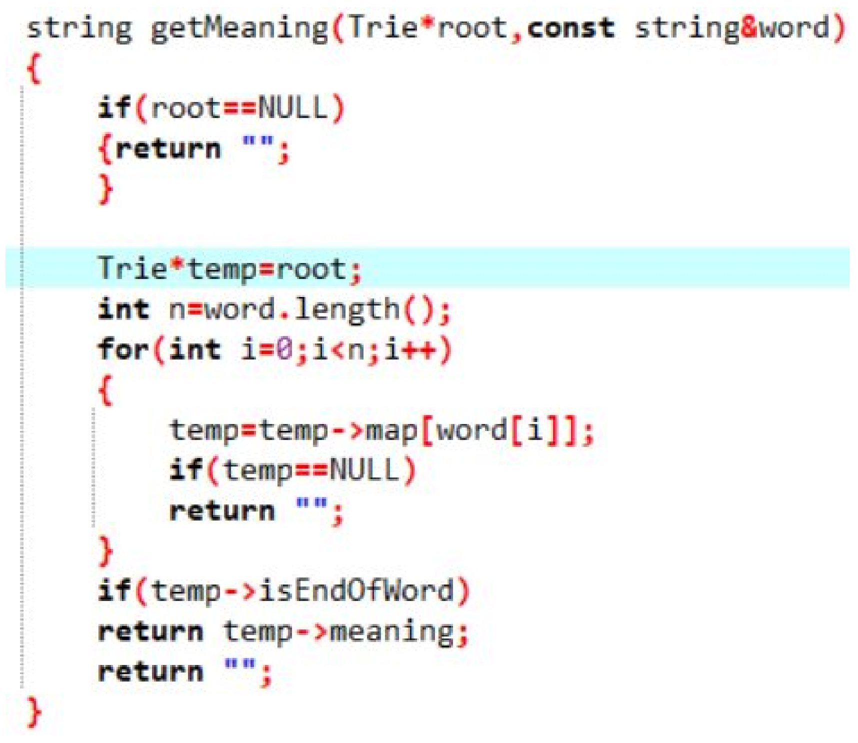
```
struct Trie{
    bool isEndOfWord;
    unordered_map<char,Trie*> map;
    string meaning;
};
Trie* getNewTrieNode()
{
    Trie*node=new Trie;
    node->isEndOfWord=false;
    return node;
}
```

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



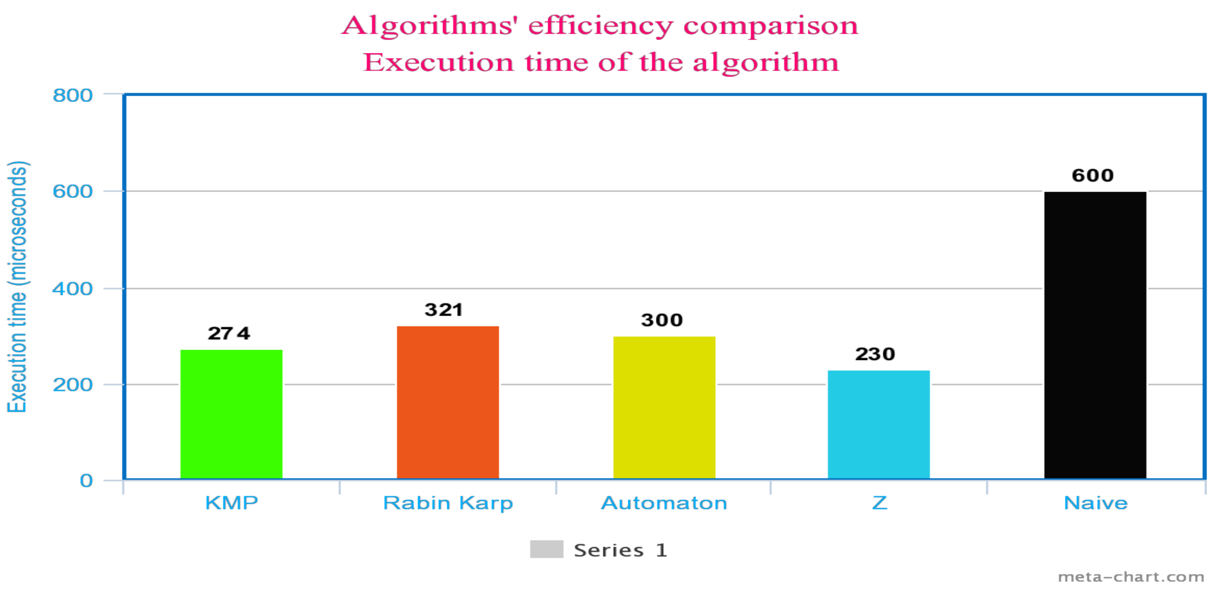
```
string getMeaning(Trie*root,const string&word)
{
    if(root==NULL)
    {return "";
    }

    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

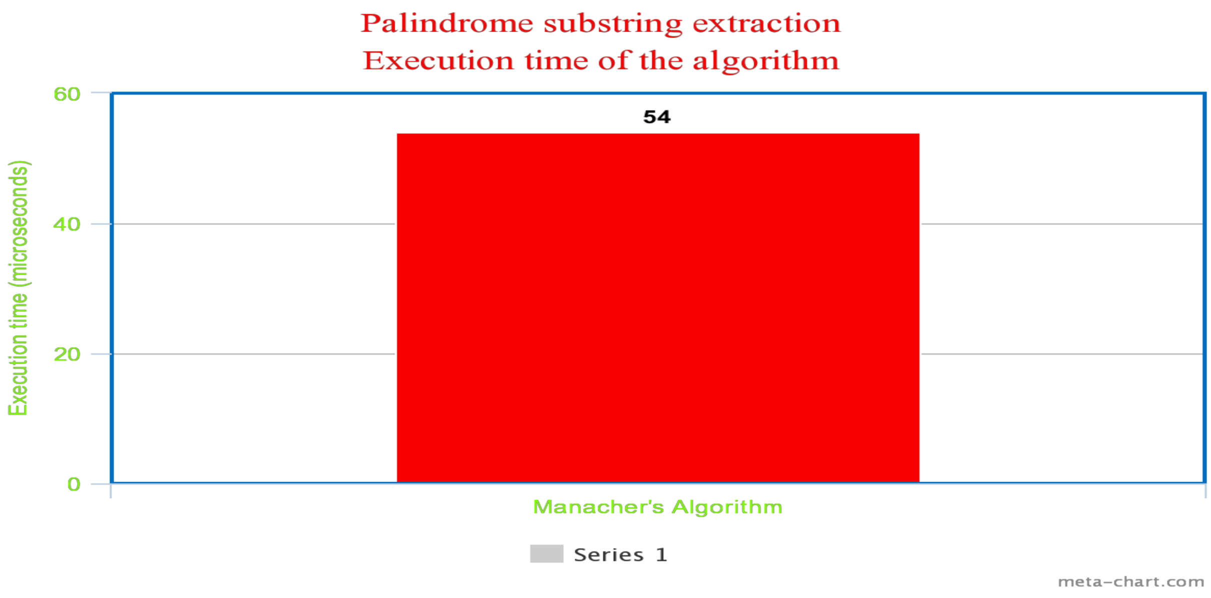
This algorithm has a worst-case time complexity of O($\Sigma|s_i|$) since to insert each character we need to perform a constant amount of work. So total work is equal to O(sum of all characters).

Time taken by different pattern matching algorithms

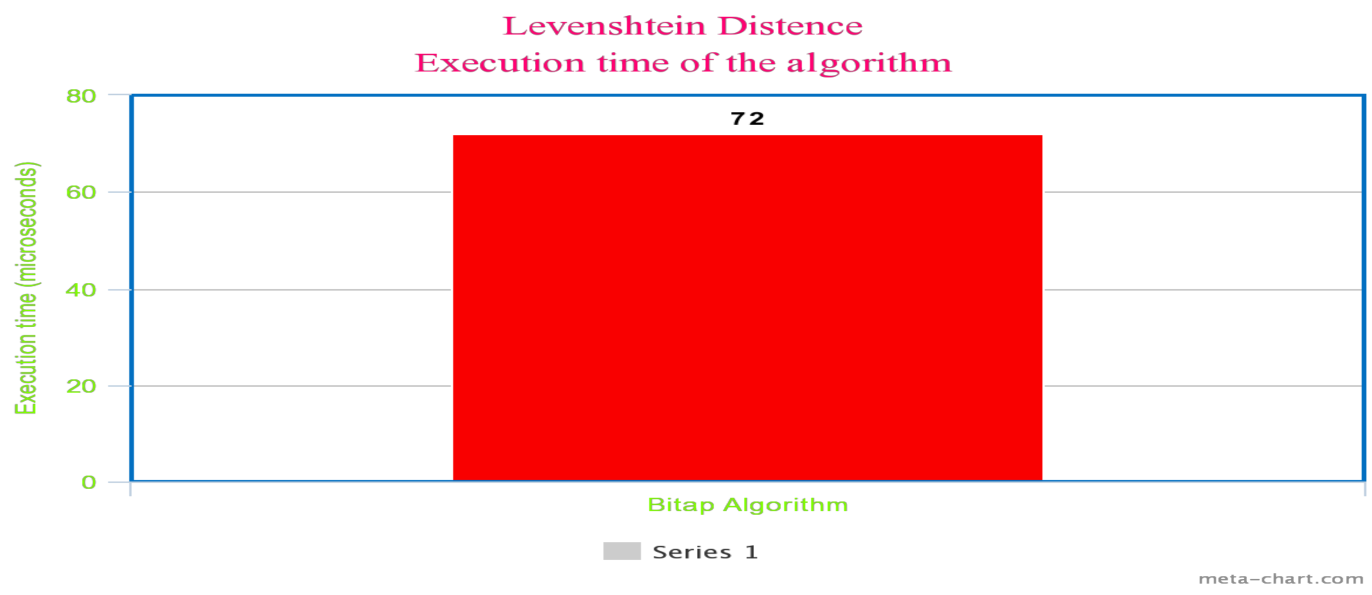


As we can see from this bar chart, we can say that KMP linear time and the algorithms derived from it have good execution time whereas Rabin Karp has got not so better execution time as it is based on window wise rolling hash.So, we can say that KMP,automaton and Z algorithms have better execution times consistently as their execution times do not depend on the nature of the data unlike Boyer Moore and Rabin Karp.

Time taken by Manacher's Algorithm



Time taken by Bitap Algorithm



Test case for string matching algorithm

Test Case used is provided here

Taken from here.

Test case for Palindromic substring

Test Case used is provided here.

Test case for Bitap Algorithm

Test Case used is:

1. **"Lucknow"**
2. **"Ahmedabad"**

References

1. ***https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf***

2. ***https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8703383***

3. *https://core.ac.uk/download/pdf/26834073.pdf*

4. *https://www.sciencedirect.com/science/article/pii/S0304397508008852*

5. **Wikipedia**

6. **Geeks for Geeks**