Pattern Searching

1. Searching for Patterns | Set 1 (Naive Pattern Searching)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that *n* > *m*.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
Pattern found at index 10
```

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
 Pattern found at index 0
 Pattern found at index 9
 Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

Naive Pattern Searching:
Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

```c
#include<stdio.h>
#include<string.h>
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }
        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        {
           printf("Pattern found at index %d \n", i);
        }
    }
}

/* Driver program to test above function */
int main()
{
   char *txt = "AABAACAADAABAAABAA";
   char *pat = "AABA";
   search(pat, txt);
   getchar();
   return 0;
}
```

What is the best case?

The best case occurs when the first character of the pattern is not present in text at all.

```
txt[]  = "AABCCAADDEE"
pat[] = "FAA"
```

The number of comparisons in best case is O(n).

What is the worst case ?

The worst case of Naive Pattern Searching occurs in following scenarios.

1) When all characters of the text and pattern are same.

```
txt[] = "AAAAAAAAAAAAAAAAAA"
pat[] = "AAAAA".
```

2) Worst case also occurs when only the last character is different.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"
```

Number of comparisons in worst case is O(m*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the

worst case to O(n). We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

2. Searching for Patterns | Set 2 (KMP Algorithm)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that *n* > *m*.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
 Pattern found at index 10
```

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
 Pattern found at index 0
 Pattern found at index 9
 Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed Naive pattern searching algorithm in the previous post. The worst case complexity of Naive algorithm is O(m(n-m+1)). Time complexity of KMP algorithm is O(n) in worst case.

KMP (Knuth Morris Pratt) Pattern Searching
The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We take advantage of this information to avoid matching the characters that we know will anyway match.
KMP algorithm does some preprocessing over the pattern pat[] and constructs an auxiliary array lps[] of size m (same as size of pattern). Here **name lps indicates longest proper prefix which is also suffix.**. For each sub-pattern pat[0…i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

```
lps[i] = the longest proper prefix of pat[0..i]
           which is also a suffix of pat[0..i].
```

Examples:
For the pattern "AABAACAABAA", lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
For the pattern "ABCDE", lps[] is [0, 0, 0, 0, 0]
For the pattern "AAAAA", lps[] is [0, 1, 2, 3, 4]
For the pattern "AAABAAA", lps[] is [0, 1, 2, 0, 1, 2, 3]
For the pattern "AAACAAAAAC", lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

Searching Algorithm:
Unlike the Naive algo where we slide the pattern by one, we use a value from lps[] to decide the next sliding position. Let us see how we do that. When we compare pat[j] with txt[i] and see a mismatch, we know that characters pat[0..j-1] match with txt[i-j+1…i-1], and we also know that lps[j-1] characters of pat[0…j-1] are both proper prefix and suffix which means we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. See KMPSearch() in the below code for details.

Preprocessing Algorithm:
In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]. See computeLPSArray () in the below code for details.

```
#include<stdio.h>
```

```c
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix values fo
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
      }

      // mismatch after j matches
      else if (i < N && pat[j] != txt[i])
      {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
         j = lps[j-1];
        else
         i = i+1;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // lenght of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
```

```
            i++;
        }
        else // (pat[i] != pat[len])
        {
          if (len != 0)
          {
            // This is tricky. Consider the example AAACAAAA and i = 7.
            len = lps[len-1];

            // Also, note that we do not increment i here
          }
          else // if (len == 0)
          {
            lps[i] = 0;
            i++;
          }
        }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

3. Searching for Patterns | Set 3 (Rabin-Karp Algorithm)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

Pattern found at index 10

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
```

```
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.
Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.
2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.
Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])* and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of “122” will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash(txt[s+1 .. s+m]) = d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q*

hash(txt[s .. s+m-1]) : Hash value at shift *s*.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift *s*+1)
d: Number of characters in the alphabet
q: A prime number
h: d^(m-1)

```
/* Following program is a C implementation of the Rabin Karp Algorithm
   given in the CLRS book */
```

```c
#include<stdio.h>
#include<string.h>

// d is the number of characters in input alphabet
#define d 256

/*  pat  -> pattern
    txt  -> text
    q    -> A prime number
*/
void search(char *pat, char *txt, int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0;  // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M-1; i++)
        h = (h*d)%q;

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < M; i++)
    {
        p = (d*p + pat[i])%q;
        t = (d*t + txt[i])%q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++)
    {

        // Check the hash values of current window of text and pattern
        // If the hash values match then only check for characters on
        if ( p == t )
        {
            /* Check for characters one by one */
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }
            if (j == M)  // if p == t and pat[0...M-1] = txt[i, i+1, .
            {
                printf("Pattern found at index %d \n", i);
            }
        }

        // Calculate hash value for next window of text: Remove leadin
        // add trailing digit
        if ( i < N-M )
        {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;

            // We might get negative value of t, converting it to posi
            if(t < 0)
              t = (t + q);
        }
    }
```

```
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEK";
    int q = 101;  // A prime number
    search(pat, txt, q);
    getchar();
    return 0;
}
```

The average and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = “AAA” and txt[] = “AAAAAAA”.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap34.htm

http://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf

http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm

Related Posts:
Searching for Patterns | Set 1 (Naive Pattern Searching)
Searching for Patterns | Set 2 (KMP Algorithm)

4. Searching for Patterns | Set 4 (A Naive Pattern Searching Question)

Question: We have discussed Naive String matching algorithm here. Consider a situation where all characters of pattern are different. Can we modify the original Naive String Matching algorithm so that it works better for these types of patterns. If we can, then what are the changes to original algorithm?

Solution: In the original Naive String matching algorithm , we always slide the pattern by 1. When all characters of pattern are different, we can slide the pattern by more than 1. Let us see how can we do this. When a mismatch occurs after j matches, we know that the first character of pattern will not match the j matched characters because all

characters of pattern are different. So we can always slide the pattern by j without missing any valid shifts. Following is the modified code that is optimized for the special patterns.

```
#include<stdio.h>
#include<string.h>

/* A modified Naive Pettern Searching algorithn that is optimized
   for the cases when all characters of pattern are different */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i = 0;

    while(i <= N - M)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }
        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        {
           printf("Pattern found at index %d \n", i);
           i = i + M;
        }
        else if (j == 0)
        {
           i = i + 1;
        }
        else
        {
           i = i + j; // slide the pattern by j
        }
    }
}

/* Driver program to test above function */
int main()
{
   char *txt = "ABCEABCDABCEABCD";
   char *pat = "ABCD";
   search(pat, txt);
   getchar();
   return 0;
}
```

Output:
Pattern found at index 4
Pattern found at index 12

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

5. Searching for Patterns | Set 5 (Finite Automata)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

Pattern found at index 10

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
 Pattern found at index 0
 Pattern found at index 9
 Pattern found at index 13
```

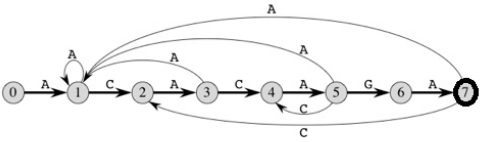
Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm

In this post, we will discuss Finite Automata (FA) based pattern searching algorithm. In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to new state. If we reach final state, then pattern is found in text. Time complexity of the search prcess is

O(n).
Before we discuss FA construction, let us take a look at the following FA for pattern ACACAGA.



state	character A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The abvoe diagrams represent graphical and tabular representations of pattern ACACAGA.

Number of states in FA will be M+1 where M is length of the pattern. The main thing to construct FA is to get the next state from the current state for every possible character. Given a character x and a state k, we can get the next state by considering the string “pat[0..k-1]x” which is basically concatenation of pattern characters pat[0], pat[1] … pat[k-1] and the character x. The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of “pat[0..k-1]x”. The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character ‘C’ in the above diagram. We need to consider the string, “pat[0..5]C” which is “ACACAC”. The lenght of the longest prefix of the pattern such that the prefix is suffix of “ACACAC”is 4 (“ACAC”). So the next state (from state 5) is 4 for character ‘C’.

In the following code, computeTF() constructs the FA. The time complexity of the computeTF() is O(m^3*NO_OF_CHARS) where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible that can be a suffix of “pat[0..k-1]x”. There are better implementations to construct FA in O(m*NO_OF_CHARS) (Hint: we can use something like lps array construction in KMP algorithm). We have covered the better implementation in our next post on pattern searching.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
    // If the character c is same as next character in pattern,
    // then simply increment state
    if (state < M && x == pat[state])
        return state+1;
```

```
        return state+1;

    int ns, i;  // ns stores the result which is next state

    // ns finally contains the longest prefix which is also suffix
    // in "pat[0..state-1]c"

    // Start from the largest possible value and stop when you find
    // a prefix which is also suffix
    for (ns = state; ns > 0; ns--)
    {
        if(pat[ns-1] == x)
        {
            for(i = 0; i < ns-1; i++)
            {
                if (pat[i] != pat[state-ns+1+i])
                    break;
            }
            if (i == ns-1)
                return ns;
        }
    }

    return 0;
}

/* This function builds the TF table which represents Finite Automata
   given pattern  */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
           TF[state][x] = getNextState(pat, M,  state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
       state = TF[state][txt[i]];
       if (state == M)
       {
         printf ("\n patterb found at index %d", i-M+1);
       }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "AABAACAADAABAAABAA";
```

```
    char *pat = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:

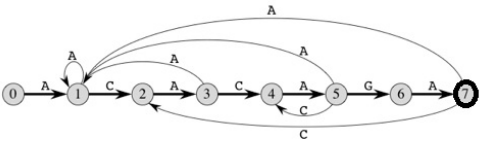
```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

References:
Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

6. Pattern Searching | Set 6 (Efficient Construction of Finite Automata)

In the previous post, we discussed Finite Automata based pattern searching algorithm. The FA (Finite Automata) construction method discussed in previous post takes O((m^3)*NO_OF_CHARS) time. FA can be constructed in O(m*NO_OF_CHARS) time. In this post, we will discuss the O(m*NO_OF_CHARS) algorithm for FA construction. The idea is similar to lps (longest prefix suffix) array construction discussed in the KMP algorithm. We use previously filled rows to fill a new row.



state	character A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The abvoe diagrams represent graphical and tabular representations of pattern ACACAGA.

Algorithm:

1) Fill the first row. All entries in first row are always 0 except the entry for pat[0] character. For pat[0] character, we always need to go to state 1.
2) Initialize lps as 0. lps for the first index is always 0.
3) Do following for rows at index i = 1 to M. (M is the length of the pattern)
…..a) Copy the entries from the row at index equal to lps.
…..b) Update the entry for pat[i] character to i+1.
…..c) Update lps "lps = TF[lps][pat[i]]" where TF is the 2D array which is being constructed.

Implementation

Following is C implementation for the above algorithm.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

/* This function builds the TF table which represents Finite Automata
   given pattern  */
void computeTransFun(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int i, lps = 0, x;

    // Fill entries in first row
    for (x =0; x < NO_OF_CHARS; x++)
       TF[0][x] = 0;
    TF[0][pat[0]] = 1;

    // Fill entries in other rows
    for (i = 1; i<= M; i++)
    {
        // Copy values from row at index lps
        for (x = 0; x < NO_OF_CHARS; x++)
            TF[i][x] = TF[lps][x];

        // Update the entry corresponding to this character
        TF[i][pat[i]] = i + 1;

        // Update lps for next row to be filled
        if (i < M)
          lps = TF[lps][pat[i]];
    }
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTransFun(pat, M, TF);

    // process text over FA.
    int i, j=0;
    for (i = 0; i < N; i++)
```

```
    {
        j = TF[j][txt[i]];
        if (j == M)
        {
          printf ("\n pattern found at index %d", i-M+1);
        }
    }
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEKS";
    search(pat, txt);
    getchar();
    return 0;
}
```

Output:

```
pattern found at index 0
pattern found at index 10
```

Time Complexity for FA construction is O(M*NO_OF_CHARS). The code for search is same as the previous post and time complexity for it is O(n).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

7. Pattern Searching | Set 7 (Boyer Moore Algorithm – Bad Character Heuristic)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Examples:

```
1) Input:

  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"

Output:

Pattern found at index 10
```

```
2) Input:

  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"

Output:

   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm
Finite Automata based Algorithm

In this post, we will discuss Boyer Moore pattern searching algorithm. Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern.
Boyer Moore is a combination of following two approaches.
1) Bad Character Heuristic
2) Good Suffix Heuristic
Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preporcesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of pattern is called the Bad Character. Whenever a character doesn't match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last

occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

```
/* Program for Bad Character Heuristic of Boyer Moore String Matching A

# include <limits.h>
# include <string.h>
# include <stdio.h>

# define NO_OF_CHARS 256

// A utility function to get maximum of two integers
int max (int a, int b) { return (a > b)? a: b; }

// The preprocessing function for Boyer Moore's bad character heuristi
void badCharHeuristic( char *str, int size, int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
         badchar[i] = -1;

    // Fill the actual value of last occurrence of a character
    for (i = 0; i < size; i++)
         badchar[(int) str[i]] = i;
}
```

```
/* A pattern searching function that uses Bad Character Heuristic of
   Boyer Moore Algorithm */
void search( char *txt,  char *pat)
{
    int m = strlen(pat);
    int n = strlen(txt);

    int badchar[NO_OF_CHARS];

    /* Fill the bad character array by calling the preprocessing
       function badCharHeuristic() for given pattern */
    badCharHeuristic(pat, m, badchar);

    int s = 0;  // s is shift of the pattern with respect to text
    while(s <= (n - m))
    {
        int j = m-1;

        /* Keep reducing index j of pattern while characters of
           pattern and text are matching at this shift s */
        while(j >= 0 && pat[j] == txt[s+j])
            j--;

        /* If the pattern is present at current shift, then index j
           will become -1 after the above loop */
        if (j < 0)
        {
            printf("\n pattern occurs at shift = %d", s);

            /* Shift the pattern so that the next character in text
               aligns with the last occurrence of it in pattern.
```

```
                The condition s+m < n is necessary for the case when
                pattern occurs at the end of text */
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;

        }

        else
            /* Shift the pattern so that the bad character in text
               aligns with the last occurrence of it in pattern. The
               max function is used to make sure that we get a positiv
               shift. We may get a negative shift if the last occurren
               of bad character in pattern is on the right side of the
               current character. */
            s += max(1, j - badchar[txt[s+j]]);
    }
}

/* Driver program to test above funtion */
int main()
{
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;
}
```

Output:

```
pattern occurs at shift = 4
```

The Bad Character Heuristic may take O(mn) time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

8. Pattern Searching | Set 8 (Suffix Tree Introduction)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Preprocess Pattern or Preoprocess Text?
We have discussed the following algorithms in the previous posts:

KMP Algorithm
Rabin Karp Algorithm
Finite Automata based Algorithm

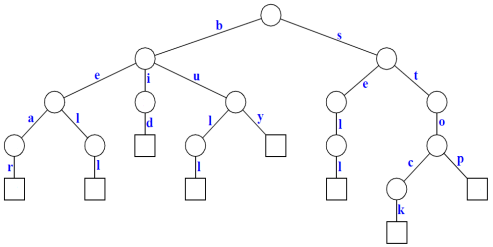
Boyer Moore Algorithm

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is O(n) where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in O(m) time where m is length of the pattern.
Imagine you have stored complete work of William Shakespeare and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.
Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

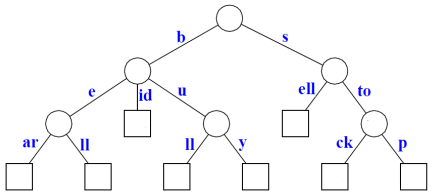
A Suffix Tree for a given text is a compressed trie for all suffixes of the given text.
We have discussed Standard Trie. Let us understand **Compressed Trie** with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.

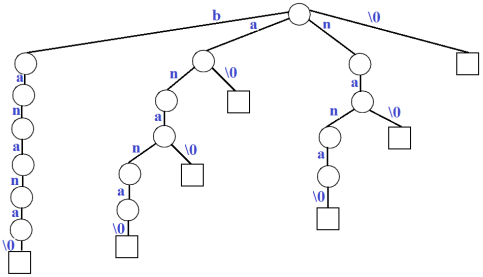


How to build a Suffix Tree for a given text?
As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.
1) Generate all suffixes of given text.
2) Consider all suffixes as individual words and build a compressed trie.

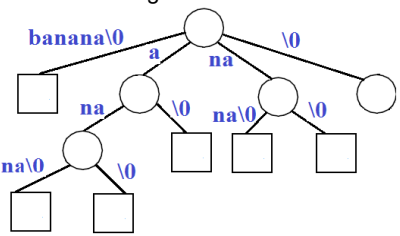
Let us consider an example text “banana\0″ where ‘\0′ is string termination character. Following are all suffixes of “banana\0″

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text “banana\0″

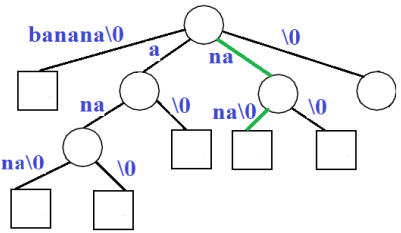


Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?
We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.
1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
…..**a)** For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.
…..**b)** If there is no edge, print “pattern doesn’t exist in text” and return.

2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print “Pattern found”.

Let us consider the example pattern as “nan” to see the searching process. Following diagram shows the path followed for searching “nan” or “nana”.



How does this work?
Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree
Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.
1) Pattern Searching
2) Finding the longest repeated substring
3) Finding the longest common substring
4) Finding the longest palindrome in a string

There are many more applications. See this for more details.

Ukkonen’s Suffix Tree Construction is discussed in following articles:
Ukkonen’s Suffix Tree Construction – Part 1
Ukkonen’s Suffix Tree Construction – Part 2
Ukkonen’s Suffix Tree Construction – Part 3
Ukkonen’s Suffix Tree Construction – Part 4
Ukkonen’s Suffix Tree Construction – Part 5
Ukkonen’s Suffix Tree Construction – Part 6

References:
http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf
http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf
http://www.allisons.org/ll/AlgDS/Tree/Suffix/

9. String matching where one string contains wildcard characters

Given two strings where first string may contain wild card characters and second string is a normal string. Write a function that returns true if the two strings match. The following are allowed wild card characters in first string.

```
* --> Matches with 0 or more instances of any character or set of characters.
? --> Matches with any one character.
```

For example, “g*ks” matches with “geeks” match. And string “ge?ks*” matches with “geeksforgeeks” (note ‘*’ at the end of first string). But “g*k” doesn’t match with “gee” as character ‘k’ is not present in second string.

```c
// A C program to match wild card characters
#include <stdio.h>
#include <stdbool.h>

// The main function that checks if two given strings match. The first
// string may contain wildcard characters
bool match(char *first, char * second)
{
    // If we reach at the end of both strings, we are done
    if (*first == '\0' && *second == '\0')
        return true;

    // Make sure that the characters after '*' are present in second s
    // This function assumes that the first string will not contain tw
    // consecutive '*'
    if (*first == '*' && *(first+1) != '\0' && *second == '\0')
        return false;

    // If the first string contains '?', or current characters of both
    // strings match
    if (*first == '?' || *first == *second)
        return match(first+1, second+1);

    // If there is *, then there are two possibilities
    // a) We consider current character of second string
    // b) We ignore current character of second string.
    if (*first == '*')
        return match(first+1, second) || match(first, second+1);
    return false;
}

// A function to run test cases
void test(char *first, char *second)
{  match(first, second)? puts("Yes"): puts("No"); }

// Driver program to test above functions
int main()
{
    test("g*ks", "geeks"); // Yes
    test("ge?ks*", "geeksforgeeks"); // Yes
    test("g*k", "gee");  // No because 'k' is not in second
    test("*pqrs", "pqrst"); // No because 't' is not in first
    test("abc*bcd", "abcdhghgbcd"); // Yes
    test("abc*c?d", "abcd"); // No because second must have 2 instance
    test("*c*d", "abcd"); // Yes
    test("*?c*d", "abcd"); // Yes
    return 0;
}
```

Output:

```
Yes
Yes
No
No
Yes
No
```

```
Yes
Yes
```

Exercise
1) In the above solution, all non-wild characters of first string must be there is second string and all characters of second string must match with either a normal character or wildcard character of first string. Extend the above solution to work like other pattern searching solutions where the first string is pattern and second string is text and we should print all occurrences of first string in second.

2) Write a pattern searching function where the meaning of '?' is same, but '*' means 0 or more occurrences of the character just before '*'. For example, if first string is 'a*b', then it matches with 'aaab', but doesn't match with 'abb'.

This article is compiled by Vishal Chaudhary and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

10. Suffix Array | Set 1 (Introduction)

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

Pattern Searching | Set 8 (Suffix Tree Introduction)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to Suffix Tree which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source Wiki).
A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.
Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: Wiki)

Example:

```
Let the given string be "banana".

0 banana                          5 a
```

```
1 anana      Sort the Suffixes      3 ana
2 nana       ---------------->      1 anana
3 ana         alphabetically        0 banana
4 na                                4 na
5 a                                 2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

Naive method to build Suffix Array

A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return suffixArr;
```

```
    return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt,  n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
Following is suffix array for banana
5 3 1 0 4 2
```

The time complexity of above method to build suffix array is $O(n^2Logn)$ if we consider a O(nLogn) algorithm used for sorting. The sorting step itself takes $O(n^2Logn)$ time as every comparison is a comparison of two strings and the comparison takes O(n) time. There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, Binary Search can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```
// This code only contains search() and main. To make it a complete ru
// above code or see http://ideone.com/1Io9eN

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat);  // get length of pattern, needed for strncmp

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1;  // Initilize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strncmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabtically less than
        // the mid suffix
        if (res < 0) r = mid - 1;

        // Otherwise move to right half
        else l = mid + 1;
    }

    // We reach here if return statement in loop is not executed
    cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana";  // text
    char pat[] = "nan";   // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}
```

Output:

```
Pattern found at index 2
```

The time complexity of the above search function is O(mLogn). There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a O(m) suffix

array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array
Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.
1) Pattern Searching
2) Finding the longest repeated substring
3) Finding the longest common substring
4) Finding the longest palindrome in a string

See this for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed a O(nLogn) algorithm for Suffix Array construction here. We will soon be discussing more efficient suffix array algorithms.

References:
http://www.stanford.edu/class/cs97si/suffix-array.pdf
http://en.wikipedia.org/wiki/Suffix_array

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

11. Anagram Substring Search (Or Search for all permutations)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] and its permutations (or anagrams) in txt[]. You may assume that n > m.
Expected time complexity is O(n)

Examples:

```
1) Input:  txt[] = "BACDGABCDA"  pat[] = "ABCD"
   Output:   Found at Index 0
             Found at Index 5
             Found at Index 6
2) Input: txt[] =  "AAABABAA" pat[] = "AABA"
   Output:   Found at Index 0
             Found at Index 1
             Found at Index 4
```

This problem is slightly different from standard pattern searching problem, here we need

to search for anagrams as well. Therefore, we cannot directly apply standard pattern searching algorithms like KMP, Rabin Karp, Boyer Moore, etc.

A simple idea is to modify Rabin Karp Algorithm. For example we can keep the hash value as sum of ASCII values of all characters under modulo of a big prime number. For every character of text, we can add the current character to hash value and subtract the first character of previous window. This solution looks good, but like standard Rabin Karp, the worst case time complexity of this solution is O(mn). The worst case occurs when all hash values match and we one by one match all characters.

We can achieve O(n) time complexity under the assumption that alphabet size is fixed which is typically true as we have maximum 256 possible characters in ASCII. The idea is to use two count arrays:

1) The first count array store frequencies of characters in pattern.
2) The second count array stores frequencies of characters in current window of text.

The important thing to note is, time complexity to compare two count arrays is O(1) as the number of elements in them are fixed (independent of pattern and text sizes). Following are steps of this algorithm.
1) Store counts of frequencies of pattern in first count array *countP[]*. Also store counts of frequencies of characters in first window of text in array *countTW[]*.

2) Now run a loop from i = M to N-1. Do following in loop.
…..a) If the two count arrays are identical, we found an occurrence.
…..b) Increment count of current character of text in countTW[]
…..c) Decrement count of first character in previous window in countWT[]

3) The last window is not checked by above loop, so explicitly check it.

Following is C++ implementation of above algorithm.

```
// C++ program to search all anagrams of a pattern in a text
#include<iostream>
#include<cstring>
#define MAX 256
using namespace std;

// This function returns true if contents of arr1[] and arr2[]
// are same, otherwise false.
bool compare(char arr1[], char arr2[])
{
    for (int i=0; i<MAX; i++)
        if (arr1[i] != arr2[i])
            return false;
    return true;
}

// This function search for all permutations of pat[] in txt[]
void search(char *pat, char *txt)
{
    int M = strlen(pat), N = strlen(txt);

    // countP[]:  Store count of all characters of pattern
    // countTW[]: Store count of current window of text
    char countP[MAX] = {0}, countTW[MAX] = {0};
    for (int i = 0; i < M; i++)
    {
        (countP[pat[i]])++;
        (countTW[txt[i]])++;
    }

    // Traverse through remaining characters of pattern
    for (int i = M; i < N; i++)
    {
        // Compare counts of current window of text with
        // counts of pattern[]
        if (compare(countP, countTW))
            cout << "Found at Index " << (i - M) << endl;

        // Add current character to current window
        (countTW[txt[i]])++;

        // Remove the first character of previous window
        countTW[txt[i-M]]--;
    }

    // Check for the last window in text
    if (compare(countP, countTW))
        cout << "Found at Index " << (N - M) << endl;
}

/* Driver program to test above function */
int main()
{
    char txt[] = "BACDGABCDA";
    char pat[] = "ABCD";
    search(pat, txt);
    return 0;
}
```

Output:

```
Found at Index 0
Found at Index 5
Found at Index 6
```

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

12. Pattern Searching using a Trie of all Suffixes

Problem Statement: Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

As discussed in the previous post, we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: KMP Algorithm, Rabin Karp Algorithm, Finite Automata, Boyer Moore Algorithm.

2) Preoprocess Text: Suffix Tree

The best possible time complexity achieved by first (preprocssing pattern) is O(n) and by second (preprocessing text) is O(m) where m and n are lengths of pattern and text respectively.

Note that the second way does the searching only in O(m) time and it is preferred when text doesn't doesn't change very frequently and there are many search queries. We have discussed Suffix Tree (A compressed Trie of all suffixes of Text) .

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a Standard Trie of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a simple Trie instead of compressed Trie.

As discussed in Suffix Tree post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in O(m) time where m is pattern length.

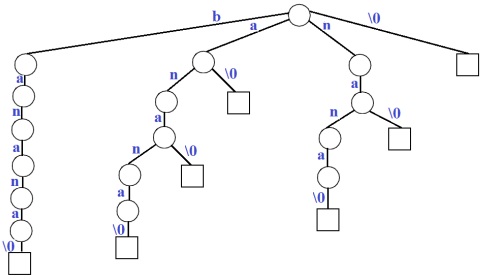
Building a Trie of Suffixes
1) Generate all suffixes of given text.
2) Consider all suffixes as individual words and build a trie.

Let us consider an example text "banana\0" where '\0' is string termination character.

Following are all suffixes of “banana\0”

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a Trie, we get following.



How to search a pattern in the built Trie?
Following are steps to search a pattern in the built Trie.
1) Starting from the first character of the pattern and root of the Trie, do following for every character.
…..**a)** For the current character of pattern, if there is an edge from the current node, follow the edge.
…..**b)** If there is no edge, print “pattern doesn’t exist in text” and return.
2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is C++ implementation of the above idea.

```
// A simple C++ implementation of substring search using trie of suffi
#include<iostream>
#include<list>
#define MAX_CHAR 256
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTrieNode
{
private:
    SuffixTrieNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTrieNode() // Constructor
```

```
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;

        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
          children[i] = NULL;
    }

    // A recursive function to insert a suffix of the txt
    // in subtree rooted with this node
    void insertSuffix(string suffix, int index);

    // A function to search a pattern in subtree rooted
    // with this node.The function returns pointer to a linked
    // list containing all indexes where pattern is present.
    // The returned indexes are indexes of last characters
    // of matched text.
    list<int>* search(string pat);
};

// A Trie of all suffixes
class SuffixTrie
{
private:
    SuffixTrieNode root;
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTrieNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }

    // Function to searches a pattern in this suffix trie.
    void search(string pat);
};

// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTrieNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_front(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTrieNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index+1);
```

```
    }
}

// A recursive function to search a pattern in subtree rooted with
// this node
list<int>* SuffixTrieNode::search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;

    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)])->search(s.substr(1));

    // If there is no edge, pattern doesn't exist in text
    else return NULL;
}

/* Prints all occurrences of pat in the Suffix Trie S (built for text)
void SuffixTrie::search(string pat)
{
    // Let us call recursive search function for root of Trie.
    // We get a list of all indexes (where pat is present in text) in
    // variable 'result'
    list<int> *result = root.search(pat);

    // Check if the list of indexes is empty or not
    if (result == NULL)
        cout << "Pattern not found" << endl;
    else
    {
       list<int>::iterator i;
       int patLen = pat.length();
       for (i = result->begin(); i != result->end(); ++i)
         cout << "Pattern found at position " << *i - patLen<< endl;
    }
}

// driver program to test above functions
int main()
{
    // Let us build a suffix trie for text "geeksforgeeks.org"
    string txt = "geeksforgeeks.org";
    SuffixTrie S(txt);

    cout << "Search for 'ee'" << endl;
    S.search("ee");

    cout << "\nSearch for 'geek'" << endl;
    S.search("geek");

    cout << "\nSearch for 'quiz'" << endl;
    S.search("quiz");

    cout << "\nSearch for 'forgeeks'" << endl;
    S.search("forgeeks");

    return 0;
}
```

Output:

```
Search for 'ee'
Pattern found at position 9
Pattern found at position 1

Search for 'geek'
Pattern found at position 8
Pattern found at position 0

Search for 'quiz'
Pattern not found

Search for 'forgeeks'
Pattern found at position 5
```

Time Complexity of the above search function is O(m+k) where m is length of the pattern and k is the number of occurrences of pattern in text.

This article is contributed by Ashish Anand. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

13. Ukkonen's Suffix Tree Construction – Part 1

Suffix Tree is very useful in numerous string processing and computational biology problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and it's not easy to implement code to construct suffix tree and it's usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.

Note: You may find some portion of the algorithm difficult to understand while 1st or 2nd reading and it's perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology by **Dan Gusfield** explains the concepts very well.

A suffix tree **T** for a m-character string S is a rooted directed tree with exactly m leaves numbered 1 to **m.** (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of S that starts at position i, i.e. S[i…m].

Note: Position starts with 1 (it's not zero indexed, but later, while code implementation, we will used zero indexed position)

For string S = xabxac with m = 6, suffix tree will look like following:

It has one root node and two internal nodes and 6 leaf nodes.

String Depth of red path is 1 and it represents suffix c starting at position 6
String Depth of blue path is 4 and it represents suffix bxca starting at position 3
String Depth of green path is 2 and it represents suffix ac starting at position 5
String Depth of orange path is 6 and it represents suffix xabxac starting at position 1

Edges with labels a (green) and xa (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of S matches a prefix of another suffix of S (when last character in not unique in string), then path for the first suffix would not end at a leaf.

For String S = xabxa, with m = 5, following is the suffix tree:

Here we will have 5 suffixes: xabxa, abxa, bxa, xa and a.
Path for suffixes 'xa' and 'a' do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes ('xa' and 'a') are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use $, # etc as termination characters.
Following is the suffix tree for string S = xabxa$ with m = 6 and now all 6 suffixes end at leaf.

A naive algorithm to build a suffix tree
Given a string S of length m, enter a single edge for suffix S[l ..m]$ (the entire string) into

the tree, then successively enter suffix S[i..m]$ into the growing tree, for i increasing from 2 to m. Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i. So N_i+1 is constructed from N_i as follows:

- Start at the root of N_i
- Find the longest path from the root which matches a prefix of S[i+1..m]$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v), break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in S[i+l..m] and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with S[i+1..m], and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge (w, i+1) from w to a new leaf labelled i+1 and it labels the new edge with the unmatched part of suffix S[i+1..m]

This takes $O(m^2)$ to build the suffix tree for the string S of length m.
Following are few steps to build suffix tree based for string “xabxa$” based on above algorithm:

Implicit suffix tree
While generating suffix tree using Ukkonen’s algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with $ (or # or any other termination character) label and no internal node with only one edge going out of it.
To get implicit suffix tree from a suffix tree S$,

- Remove all terminal symbol $ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

High Level Description of Ukkonen’s algorithm
Ukkonen’s algorithm constructs an implicit suffix tree T_i for each prefix S[l ..i] of S (of length m).
It first builds T_1 using 1^{st} character, then T_2 using 2^{nd} character, then T_3 using 3^{rd} character, …, T_m using m^{th} character.

Implicit suffix tree T_i+1 is built on top of implicit suffix tree T_i.

The true suffix tree for S is built from T_m by adding $.

At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.
Time taken is O(m).

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)
In phase i+1, tree T_i+1 is built from tree T_i.

Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1..i+1]
In extension j of phase i+1, the algorithm first finds the end of the path from the root labelled with substring S[j..i].
It then extends the substring by adding the character S(i+1) to its end (if it is not there already).
In extension 1 of phase i+1, we put string S[1..i+1] in the tree. Here S[1..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already).
In extension 2 of phase i+1, we put string S[2..i+1] in the tree. Here S[2..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
In extension 3 of phase i+1, we put string S[3..i+1] in the tree. Here S[3..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
.
.
In extension i+1 of phase i+1, we put string S[i+1..i+1] in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label S[i+1].

High Level Ukkonen's algorithm

```
Construct tree T1
For i from 1 to m-1 do
begin {phase i+1}
    For j from 1 to i+1
        begin {extension j}
        Find the end of the path from the root labelled S[j..i] in the current tree.
        Extend that path by adding character S[i+l] if it is not there already
    end;
end;
```

Suffix extension is all about adding the next character into the suffix tree built so far.
In extension j of phase i+1, algorithm finds the end of S[j..i] (which is already in the tree

due to previous phase i) and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.

There are 3 extension rules:
Rule 1: If the path from the root labelled S[j..i] ends at leaf edge (i.e. S[i] is last character on leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1].
A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is s[i+1] (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string xabxac using Ukkonen's algorithm:

In next parts (Part 2, Part 3, Part 4 and Part 5), we will discuss suffix links, active points, few tricks and finally code implementations (Part 6).

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

14. Ukkonen's Suffix Tree Construction – Part 2

In Ukkonen's Suffix Tree Construction – Part 1, we have seen high level Ukkonen's Algorithm. This 2nd part is continuation of Part 1.
Please go through Part 1, before looking at current article.

In Suffix Tree Construction of string S of length m, there are m phases and for a phase j (1 <= j <= m), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules (discussed in Part 1).

To do j^{th} extension of phase i+1 (adding character S[i+1]), we first need to find end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in O(m) which we will see now.

Suffix links
For an internal node v with path-label xA, where x denotes a single character and A denotes a (possibly empty) substring, if there is another node s(v) with path-label A, then a pointer from v to s(v) is called a suffix link.
If A is empty string, suffix link from internal node will go to root node.
There will not be any suffix link from root node (As it's not considered as internal node).

In extension j of some phase i, if a new internal node v with path-label xA is added, then in extension j+1 in the same phase i:

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension j+1 of same phase i, we will create a suffix link from the internal node created in j^{th} extension to the node with path labelled A.

So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension.

In any implicit suffix tree T_i after phase i, if internal node v has path-label xA, then there is a node s(v) in T_i with path-label A and node v will point to node s(v) using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

How suffix links are used to speed up the implementation?

In extension j of phase i+1, we need to find the end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. Suffix links provide a short cut to find end of the path.

So we can see that, to find end of path S[j..i], we need not traverse from root. We can start from the end of path S[j-1..i], walk up one edge to node v (i.e. go to parent node), follow the suffix link to s(v), then walk down the path y (which is abcd here in Figure 17). This shows the use of suffix link is an improvement over the process.
Note: In the next part 3, we will introduce activePoint which will help to avoid “walk up”. We can directly go to node s(v) from node v.

When there is a suffix link from node v to node s(v), then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node s(v) to a leaf. In Figure 17, there is a path label “abcd” from node v to a leaf, then there is a path will same label “abcd” from node s(v) to a leaf.
This fact can be used to improve the walk from s(v) to leaf along the path y. This is called “skip/count” trick.

Skip/Count Trick
When walking down from node s(v) to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.
If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are m phases and each phase takes O(m).

Edge-label compression
So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S. With this, suffix tree needs O(m) space.

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick “skip/count” is seen already while walk down).

Observation 1: Rule 3 is show stopper

In a phase i, there are i extensions (1 to i) to be done.

When rule 3 applies in any extension j of phase i+1 (i.e. path labelled S[j..i] continues with character S[i+1]), then it will also apply in all further extensions of same phase (i.e. extensions j+1 to i+1 in phase i+1). That’s because if path labelled S[j..i] continues with character S[i+1], then path labelled S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also continue with character S[i+1].

Consider Figure 11, Figure12 and Figure 13 in Part 1 where Rule 3 is applied.

In Figure 11, “xab” is added in tree and in Figure 12 (Phase 4), we add next character “x”. In this, 3 extensions are done (which adds 3 suffixes). Last suffix “x” is already present in tree.

In Figure 13, we add character “a” in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes “xa” and “a” are already present in tree. This shows that if suffix S[j..i] present in tree, then ALL the remaining suffixes S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also be there in tree and no work needed to add those remaining suffixes.

So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension j+1, then we need to add suffix link from node v to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

Trick 2

Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf

Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j, extension rule 1 will always apply to extension j in all successive phases.

Consider Figure 9 to Figure 14 in Part 1.

In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i, there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.

Also rule 2 creates a new leaf always (and internal node sometimes).

If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th}

phase, there will be J_i leaves labelled 1, 2, 3, …, J_i) , then $J_i <= J_{i+1}$

J_i will be equal to J_{i+1} when there are no new leaf created in phase i+1 (i.e rule 3 is applied in J_{i+1} extension)

In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$

In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$

In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$

J_i will be less than J_{i+1} when few new leaves are created in phase i+1.

In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase i+1, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.

Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i, end = i for leaf edges, for phase i+1, end = i+1 for leaf edges.

Trick 3

In any phase i, leaf edges may look like (p, i), (q, i), (r, i), …. where p, q, r are starting position of different edges and i is end position of all. Then in phase i+1, these leaf edges will look like (p, i+1), (q, i+1), (r, i+1),…. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e), (q, e), (r, e).. In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree Tm could be implicit tree if a suffix is prefix of another. So we can add a $ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next Part 3, we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints.

We will continue to discuss the algorithm in Part 4 and Part 5. Code implementation will be discussed in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

15. Ukkonen's Suffix Tree Construction – Part 3

This article is continuation of following two articles:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2

Please go through Part 1 and Part 2, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks.

Here we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree.
We will add $ (discussed in Part 1 why we do this) so string S would be "abcabxabcd$".

While building suffix tree for string S of length m:

- There will be m phases 1 to m (one phase for each character)
  In our current example, m is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree, ……, m^{th} phase will add m^{th} character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies) without going through all i extensions
- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being

traversed)

- Phase 1 will read first character from the string, will go through 1 extension.
  (In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in Part 2)
  Extension 1 will add suffix “a” in tree. We start from root and traverse path with label ‘a’. There is no path from root, going out with label ‘a’, so create a leaf edge (Rule 2).

  Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)
  For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions.
  In our example, phase 2 will read second character ‘b’. Suffixes to be added are “ab” and “b”.
  Extension 1 adds suffix “ab” in tree.
  Path for label ‘a’ ends at leaf edge, so add ‘b’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

  Extension 2 adds suffix “b” in tree. There is no path from root, going out with label ‘b’, so creates a leaf edge (Rule 2).

  Phase 2 completes with the completion of extension 2.
  Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions.
  In our example, phase 3 will read third character ‘c’. Suffixes to be added are “abc”, “bc” and “c”.
  Extension 1 adds suffix “abc” in tree.
  Path for label ‘ab’ ends at leaf edge, so add ‘c’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

  Extension 2 adds suffix “bc” in tree.
  Path for label ‘b’ ends at leaf edge, so add ‘c’ at the end of this edge.
  Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

Extension 3 adds suffix “c” in tree. There is no path from root, going out with label ‘c’, so creates a leaf edge (Rule 2).

Phase 3 completes with the completion of extension 3.
Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions.
  In our example, phase 4 will read fourth character ‘a’. Suffixes to be added are “abca”, “bca”, “ca” and “a”.
  Extension 1 adds suffix “abca” in tree.
  Path for label ‘abc’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 2 adds suffix “bca” in tree.
  Path for label ‘bc’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 3 adds suffix “ca” in tree.
  Path for label ‘c’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 4 adds suffix “a” in tree.
  Path for label ‘a’ exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix “a” is not seen explicitly (because it doesn’t end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.
  Phase 4 completes as soon as Rule 3 is applied while Extension 4.
  Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

1. At the end of any phase i, there are at most i leaf edges (if i^{th} character is not seen so far, there will be i leaf edges, else there will be less than i leaf edges).
   e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).
2. After completing phase i, “end” indices of all leaf edges are i. How do we implement

this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the “end” index? Answer is “NO”.
For this, we will maintain a global variable (say “END”) and we will just increment this global variable “END” and all leaf edge end indices will point to this global variable. So this way, if we have j leaf edges after phase i, then in phase i+1, first j extensions (1 to j) will be done by just incrementing variable “END” by 1 (END will be i+1 at the point).
Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the j leaf edges (i.e. extension 1 to j) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to j). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.

3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are j leaf edges after phase i, then in phase i+1, first j extensions will follow Rule 1 and will be done in constant time using trick 3. There are i+1-j extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase i is completed.

   If previous phase i went through all the i extensions (when i^{th} character is unique so far), then in next phase i+1, trick 3 will take care of first i suffixes (the i leaf edges) and then extension i+1 will start from root node and it will insert just one character [$(i+1)^{th}$] suffix in tree by creating a leaf edge using Rule 2.
   If previous phase i completes early (and this will happen if and only if rule 3 applies – when i^{th} character is already seen before), say at j^{th} extension (i.e. rule 3 is applied at j^{th} extension), then there are j-1 leaf edges so far.
   We will state few more facts (which may be a repeat, but we want to make sure it’s clear to you at this point) here based on discussion so far:
   - *Phase 1 starts with Rule 2, all other phases start with Rule 1*
   - *Any phase ends with either Rule 2 or Rule 3*
   - *Any phase i may go through a series of j extensions (1 <= j <= i). In these j extensions, first p (0 <= p < i) extensions will follow Rule 1, next q (0 <= q <= i-p) extensions will follow Rule 2 and next r (0<= r <= 1) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3*
   - *In a phase i, p + q + r <= i*
   - *At the end of any phase i, there will be p+q leaf edges and next phase i+1 will go through Rule 1 for first p+q extensions*

   In the next phase i+1, trick 3 (Rule 1) will take care of first j-1 suffixes (the j-1 leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need

to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall algorithm will not be linear. activePoint comes to the rescue here.
In previous phase i, while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase i+1 will start exactly from the same point and we start matching path against $(i+1)^{th}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1^{st} p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset activePoint as apprppriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).
To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.
activeNode: This could be root node or an internal node.
activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.
activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO.
(click on below image to see it clearly)

After phase i, if there are j leaf edges then in phase i+1, first j extensions will be

done by trick 3. activePoint will be needed for the extensions from j+1 to i+1 and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i, then before we move on to next phase i+1, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLenth is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase i+1.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is (A, s, 11) in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLenght as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:
(A, s, 11) — >>> (B, w, 7) — >>> (C, a, 3)
All above three activePoints refer to same point 'c'
Let's take another example.
If activePoint is (D, a, 11) at the start of an extension, then while walk down, below is the sequence of changes in activePoint:
(D, a, 10) — >>> (E, d, 7) — >>> (F, f, 5) — >> (F, j, 1)
All above activePoints refer to same point 'k'.
If activePoints are (A, s, 3), (A, t, 5), (B, w, 1), (D, a, 2) etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.
The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

4. While code implementation, we will loop through all the characters of string S one by

one. Each loop for i^{th} character will do processing for phase i. Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase i+1, we don't really have to perform all i+1 extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in Part 4 and Part 5. Code implementation will be discussed in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

16. Ukkonen's Suffix Tree Construction – Part 4

This article is continuation of following three articles:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3

Please go through Part 1, Part 2 and Part 3, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through four phases of building suffix tree.

Let's revisit those four phases we have seen already in Part 3, in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding, we are giving character value to activeEdge, but in code implemntation, it will be index of the character) and activeLength is ZERO.

- The global variable END and remainingSuffixCount are initialized to ZERO

**********Phase 1**********

In Phase 1, we read 1st character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
  - Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
  - Once extension is performed, decrement the remainingSuffixCount by 1
  - At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 20 in Part 3 is the resulting tree after phase 1.

**********Phase 2**********

In Phase 2, we read 2nd character (b) from string S

- Set END to 2 (This will do extension 1)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'b'). This is **APCFALZ**.
  - Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
  - Once extension is performed, decrement the remainingSuffixCount by 1
  - At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 22 in Part 3 is the resulting tree after phase 2.

**********Phase 3**********

In Phase 3, we read 3rd character (c) from string S

- Set END to 3 (This will do extensions 1 and 2)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here

activeEdge will be ‘c’). This is **APCFALZ**.
    - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
    - Once extension is performed, decrement the remainingSuffixCount by 1
    - At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 25 in Part 3 is the resulting tree after phase 3.

**********Phase 4**********

In Phase 4, we read 4th character (a) from string S

- Set END to 4 (This will do extensions 1, 2 and 3)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
    - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be ‘a’). This is **APCFALZ**.
    - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 – skip/count). In our example, edge ‘a’ is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).
    - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 4, remainingSuffixCount is 1 (One suffix ‘a’, the last one, is not added explicitly in tree, but it is there in tree implicitly).
Figure 28 in Part 3 is the resulting tree after phase 4.

Revisiting completed for 1st four phases, we will continue building the tree and see how it goes.

**********Phase 5**********

In phase 5, we read 5th character (b) from string S

- Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix “ab” and extension 5 is supposed to add suffix “b” in tree)
- Run a loop remainingSuffixCount times (i.e. two times) as below:
    - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our

example, edge ‘a’ is present going out of activeNode (i.e. root).
    - Do a walk down (The trick 1 – skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)
    - Check if current character of string S (which is ‘b’) is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
    - At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 5, remainingSuffixCount is 2 (Two suffixes, ‘ab’ and ‘b’, the last two, are not added explicitly in tree, but they are in tree implicitly).

**********Phase 6**********

In phase 6, we read 6th character (x) from string S

- Set END to 6 (This will do extensions 1, 2 and 3)

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are 3 extension left to be performed, which are extensions 4, 5 and 6 for suffixes “abx”, “bx” and “x” respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
    - While extension 4, the activePoint is (root, a, 2) which points to ‘b’ on edge starting with ‘a’.
    - In extension 4, current character ‘x’ from string S doesn’t match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
    - Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix “abx” added in tree.

Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in Part 3.

activePoint change for extension rule 2 (APCFER2):
Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set “S[i – remainingSuffixCount + 1]” where i is current phase number. Can you see why this change in activePoint? Look at current extension we just discussed above for phase 6

(i=6) again where we added suffix “abx”. There activeLength is 2 and activeEdge is ‘a’. Now in next extension, we need to add suffix “bx” in the tree, i.e. path label in next extension should start with ‘b’. So ‘b’ (the 5th character in string S) should be active edge for next extension and index of b will be “i – remainingSuffixCount + 1” (6 – 2 + 1 = 5). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.
What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**

Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in Part 2. Let’s say in phase i and extension j, suffix ‘xAabcdedg’ was added in tree. At that point, let’s say activePoint was (Node-V, a, 7), i.e. point ‘g’. So for next extension j+1, we would add suffix ‘Aabcdefg’ and for that we need to traverse 2nd path shown in Figure 18. This can be done by following suffix link from current activeNode v. Suffix link takes us to the path to be traversed somewhere in between [Node s(v)] below which the path is exactly same as how it was below the previous activeNode v. As said earlier, “activePoint gets closer to root by length 1 after every extension”, this reduction in length will happen above the node s(v) but below s(v), no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

-
  - At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
  - Next suffix to be added is ‘bx’ (with remainingSuffixCount 2).
  - Current character ‘x’ from string S doesn’t match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
    Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
  - Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix “bx” added in tree.

-

- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is 'x' (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous extension's internal node goes to root (as no new internal node created in current extension 6).
- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "x" added in tree

This completes the phase 6.
Note that phase 6 has completed all its 6 extensions (Why? Because the current character c was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters 'abcabx' read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension i, points to another internal node or root (if activeNode is root in extension i+1) by the end of extension i+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walkdown from root can be avoided.

We will go through rest of the phases (7 to 11) in Part 5 and build the tree completely and after that, we will see the code for the algorithm in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

17. Ukkonen's Suffix Tree Construction – Part 5

This article is continuation of following four articles:

Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4

Please go through Part 1, Part 2, Part 3 and Part 4, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through six phases of building suffix tree.
Here, we will go through rest of the phases (7 to 11) and build the tree completely.

**********Phase 7**********

In phase 7, we read 7th character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 6.

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix 'a')
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**. Now activePoint becomes (root, 'a', 0).
  - Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (**APCFER3**) and stop any further processing.
  - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).
Above Figure 33 is the resulting tree after phase 7.

**********Phase 8**********

In phase 8, we read 8th character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes

'ab' and 'b' respectively)

- Run a loop remainingSuffixCount times (i.e. two times) as below:
  - Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
  - Do a walk down (The trick 1 – skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
  - Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
  - At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

**********Phase 9**********

In phase 9, we read 9th character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 8.

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes 'abc', 'bc' and 'c' respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
  - Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
  - Do a walk down (The trick 1 – skip/count) if necessary. In current phase 9,walk down needed as activeLength(2) >= edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on **APCFWD** (This is first time **APCFWD** is being applied in our example).
  - Check if current character of string S (which is 'c') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (**APCFER3**) and we stop here (Rule 3).
  - At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, 'abc', 'bc' and 'c', the

last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

**********Phase 10**********

In phase 10, we read 10th character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 9.

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes 'abcd', 'bcd', 'cd' and 'd' respectively)
- Run a loop remainingSuffixCount times (i.e. four times) as below:

**********Extension 7********** Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge 'c' is present going out of activeNode (Node A). Do a walk down (The trick 1 – skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength. Check if current character of string S (which is 'd') is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with 'd' going out of activePoint, so we create a leaf edge with label 'd'. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)

The newly created internal node c (in above Figure) in current extension 7, will get it's suffix link set in next extension 8 (see Figure 38 below). Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix "abcd" added in tree. Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that's going to be 'Node B'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (Node B, c, 1).

**********Extension 8**********

Now in extension 8 (here we will add suffix 'bcd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character 'd' at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character 'd' at activePoint (Node B c, 1). So logic will be same and here we a new leaf edge with label 'd' and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

Please note the node C from previous extension (see Figure 37 above) got it's suffix link set here and node D created in current extension will get it's suffix link set in next extension. What happens if no new node created in next phase? We have seen this before in Phase 6 (Part 4) and will see again in last extension of this Phase 10. Stay Tuned. Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "bcd" added in tree. Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is 'Root Node'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (root, c, 1).

**********Extension 9**********

Now in extension 9 (here we will add suffix 'cd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

-
    - Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix "cd" added in tree.
    - Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on **APCFER2C1**, activeNode will remain 'root', activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be 'd'. So new activePoint is (root, d, 0).

**********Extension 10**********

Now in extension 10 (here we will add suffix 'd'), while adding character 'd' after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

Internal Node created in previous extension, waiting for suffix link to be set in next extension, points to root if no internal node created in next extension. In code implementation, as soon as a new internal node (Say A) gets created in an extension j, we will set it's suffix link to root node and in next extension j+1, if Rule 2 applies on an existing or newly created node (Say B), then suffix link of node A will change to the new node B , else node A will keep pointing to root Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "d" added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far)

activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them.
- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extension's activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get it's suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got it's suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).
- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

**********Phase 11**********

In phase 11, we read 11^{th} character ($) from string S

- Set END to 11 (This will do extensions 1 to 10) – because we have 10 leaf edges so far by the end of previous phase 10.

- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix '$' to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character '$' of string S being processed (**APCFALZ**).
- There is no edge going out from activeNode root, so a leaf edge with label '$' will be created (Rule 2).

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "$" added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character $ was not seen before in string S so far)

Now we have added all suffixes of string 'abcabxabcd$' in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that

number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as “stringSize – labelSize + 1”. Indexed suffix tree will look like below:

In above Figure, suffix indices are shown as character position starting with 1 (It’s not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index j (1 to m for string of length m) in above figure, in code implementation, it will be j-1 (0 to m-1)
And we are done !!!!

Data Structure to represent suffix tree
How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.
Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.
In the next Part 6, we will discuss the data structure we will use in our code implementation and the code as well.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen’s suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

18. Ukkonen’s Suffix Tree Construction – Part 6

This article is continuation of following five articles:
Ukkonen’s Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5

Please go through Part 1, Part 2, Part 3, Part 4 and Part 5, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string "abcabxabcd" where we went through all phases of building suffix tree.
Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of Part 5 article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications. There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.
- **start, end** – These two will store the edge label details from parent node to current node. (start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Lets say there are two nods A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.
- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1 .

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ?

— If A->children[c] is not NULL, there is a path, if NULL, no path.

- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;
```

```
// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;
```

```
    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                            newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
              is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to curent active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

                //APCFER3
                activeLength++;
                /*STOP all further processing in this phase
```

```
                and move on to next phase*/
                break;
            }

            /*We will be here when activePoint is in middle of
              the edge being traversed and current character
              being processed is not  on the edge (we fall off
              the tree). In this case, we add a new internal node
              and a new leaf edge going out of that new node. This
              is Extension Rule 2, where a new leaf edge and a new
            internal node get created*/
            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;

            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
```

```
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                    edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
```

```
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
//  strcpy(text, "abc"); buildSuffixTree();
//  strcpy(text, "xabxac#");    buildSuffixTree();
//  strcpy(text, "xabxa");  buildSuffixTree();
//  strcpy(text, "xabxa$"); buildSuffixTree();
    strcpy(text, "abcabxabcd$"); buildSuffixTree();
//  strcpy(text, "geeksforgeeks$"); buildSuffixTree();
//  strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
//  strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}
```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous Part 5 article):

```
$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
```

```
xabcd$ [5]
```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, KMP otherwise
- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- Linear Time Suffix Array Creation

The above basic problems can be solved by DFS traversal on suffix tree.
We will soon post articles on above problems and others like below:

- Build Generalized suffix tree
- Linear Time Longest common substring problem
- Linear Time Longest palindromic substring

And More.

Test you understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string "AABAACAADAABAAABAA$" on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3. Following are the rules applied on five consecutive extensions in some Phase i (i > 5), which ones are valid:
   A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
   B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
   C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
   D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
   E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
   F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have it's suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get it's right suffix link in next extension j+1 and get the right one in later extensions like j+2, j+3 etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1

- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

19. Suffix Tree Application 1 – Substring Check

Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

```
// A C program for substring check using Ukkonen's Suffix Tree Construc
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
```

```
    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;
```

```
    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);
```

```
        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
            and move on to next phase*/
            break;
        }

        /*We will be here when activePoint is in middle of
          the edge being traversed and current character
          being processed is not  on the edge (we fall off
          the tree). In this case, we add a new internal node
          and a new leaf edge going out of that new node. This
          is Extension Rule 2, where a new leaf edge and a new
        internal node get created*/
        splitEnd = (int*) malloc(sizeof(int));
        *splitEnd = next->start + activeLength - 1;

        //New internal node
        Node *split = newNode(next->start, splitEnd);
        activeNode->children[text[activeEdge]] = split;

        //New leaf coming out of new internal node
        split->children[text[pos]] = newNode(pos, &leafEnd);
        next->start += activeLength;
```

```
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
```

```
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                    edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}
```

```
int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res != 0)
            return res;  // match (res = 1) or no match (res = -1)
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("Pattern <%s> is a Substring\n", str);
    else
        printf("Pattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "THIS IS A TEST TEXT$");
    buildSuffixTree();

    checkForSubString("TEST");
    checkForSubString("A");
    checkForSubString(" ");
    checkForSubString("IS A");
    checkForSubString(" IS A ");
    checkForSubString("TEST1");
    checkForSubString("THIS IS GOOD");
```

```
    checkForSubString("TES");
    checkForSubString("TESA");
    checkForSubString("ISB");

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Pattern <TEST> is a Substring
Pattern <A> is a Substring
Pattern < > is a Substring
Pattern <IS A> is a Substring
Pattern < IS A > is a Substring
Pattern <TEST1> is NOT a Substring
Pattern <THIS IS GOOD> is NOT a Substring
Pattern <TES> is a Substring
Pattern <TESA> is NOT a Substring
Pattern <ISB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M.

With slight modification in traversal algorithm discussed here, we can answer following:

1. Find all occurrences of a given pattern P present in text T.
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

20. Suffix Tree Application 2 – Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application (Substring Check), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through Substring Check 1st.
In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

Lets look at following figure:

This is suffix tree for String "abcabxabcd$", showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.
Suffix Index of a path tells the index of a substring (starting from root) on that path.
Consider a path "bcd$" in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd$ are at index 7 in string.
Similarly path "bxabcd$" with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabcd, bxabcd$ are at index 4.
Similarly path "bcabxabcd$" with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcd, bcabxabcd$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring "b" is at indices 1, 4 and 7
- Substring "bc" is at indices 1 and 7

With above explanation, we should be able to see following:

- Substring "ab" is at indices 0, 3 and 6

- Substring “abc” is at indices 0 and 6
- Substring “c” is at indices 2 and 8
- Substring “xab” is at index 5
- Substring “d” is at index 9
- Substring “cd” is at index 8

…..
…..

Can you see how to find all the occurrences of a pattern in a string ?

1. 1st of all, check if the given pattern really exists in string or not (As we did in Substring Check). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don’t fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
```

```
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
```

```
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
              is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
```

```
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
            and move on to next phase*/
            break;
        }

        /*We will be here when activePoint is in middle of
          the edge being traversed and current character
          being processed is not  on the edge (we fall off
          the tree). In this case, we add a new internal node
          and a new leaf edge going out of that new node. This
          is Extension Rule 2, where a new leaf edge and a new
        internal node get created*/
        splitEnd = (int*) malloc(sizeof(int));
        *splitEnd = next->start + activeLength - 1;

        //New internal node
        Node *split = newNode(next->start, splitEnd);
        activeNode->children[text[activeEdge]] = split;

        //New leaf coming out of new internal node
        split->children[text[pos]] = newNode(pos, &leafEnd);
        next->start += activeLength;
        split->children[text[next->start]] = next;

        /*We got a new internal node here. If there is any
          internal node created in last extensions of same
          phase which is still waiting for it's suffix link
          reset, do it now.*/
        if (lastNewNode != NULL)
        {
        /*suffixLink of lastNewNode points to current newly
          created internal node*/
            lastNewNode->suffixLink = split;
        }

        /*Make the current newly created internal node waiting
          for it's suffix link reset (which is pointing to root
          at present). If we come across any other internal node
          (existing or newly created) in next extension of same
          phase, when a new leaf edge gets added (i.e. when
          Extension Rule 2 applies is any of the next extension
          of same phase) at that point, suffixLink of this node
          will point to that internal node.*/
        lastNewNode = split;
    }

    /* One suffix got added in tree, decrement the count of
      suffixes yet to be added.*/
    remainingSuffixCount--;
    if (activeNode == root && activeLength > 0) //APCFER2C1
```

```
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
```

```
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
        return 1;
    }
    int count = 0;
    int i = 0;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
```

```
        {
            count += doTraversalToCountLeaf(n->children[i]);
        }
    }
    return count;
}

int countLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res == -1)  //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("\nsubstring count: 1 and position: %d",
                                n->suffixIndex);
            else
                printf("\nsubstring count: %d", countLeaf(n));
            return 1;
        }
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
```

```
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Text: GEEKSFORGEEKS, Pattern to search: GEEKS
Found at position: 8
Found at position: 0
substring count: 2
Pattern <GEEKS> is a Substring


Text: GEEKSFORGEEKS, Pattern to search: GEEK1
Pattern <GEEK1> is NOT a Substring


Text: GEEKSFORGEEKS, Pattern to search: FOR
substring count: 1 and position: 5
Pattern <FOR> is a Substring
```

```
Text: AABAACAADAABAAABAA, Pattern to search: AABA
Found at position: 13
Found at position: 9
Found at position: 0
substring count: 3
Pattern <AABA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AA
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring


Text: AAAAAAAAA, Pattern to search: AAAA
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring


Text: AAAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
```

```
Found at position: 0
substring count: 8
Pattern <AA> is a Substring


Text: AAAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring


Text: AAAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M and then if there are Z occurrences of the pattern, it will take O(Z) to find indices of all those Z occurrences.
Overall pattern complexity is linear: O(M + Z).

A bit more detailed analysis
How many internal nodes will there in a suffix tree of string of length N ??
Answer: N-1 (Why ??)
There will be N suffixes in a string of length N.
Each suffix will have one leaf.
So a suffix tree of string of length N will have N leaves.
As each internal node has at least 2 children, an N-leaf suffix tree has at most N-1 internal nodes.
If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have Z-1 internal nodes. A tree with Z leaves can be traversed in O(Z) time.
Overall pattern complexity is linear: O(M + Z).
For a given pattern, Z (the number of occurrences) can be atmost N.
So worst case complexity can be: O(M + N) if Z is close/equal to N (A tree traversal with N nodes take O(N) time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

21. Suffix Tree Application 3 – Longest Repeated Substring

Given a text string, find Longest Repeated Substring in the text. If there are more than one Longest Repeated Substrings, get any one of them.

```
Longest Repeated Substring in GEEKSFORGEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAAAA is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabpqpq is: ab (pq is another LRS here)
```

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3rd Suffix Tree Application). Other approaches will be discussed soon.

As a prerequisite, we must know how to build a suffix tree in one or the other way. Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

Lets look at following figure:

This is suffix tree for string “ABABABA$”.
In this string, following substrings are repeated:
A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA
And Longest Repeated Substring is ABABA.
In a suffix tree, one node can’t have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).
In above figure, we can see that

- Path with Substring “A” has three internal nodes down the tree
- Path with Substring “AB” has two internal nodes down the tree
- Path with Substring “ABA” has two internal nodes down the tree
- Path with Substring “ABAB” has one internal node down the tree
- Path with Substring “ABABA” has one internal node down the tree
- Path with Substring “B” has two internal nodes down the tree
- Path with Substring “BA” has two internal nodes down the tree
- Path with Substring “BAB” has one internal node down the tree
- Path with Substring “BABA” has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??
We can see in figure that, longest repeated substring will end at the internal node which is farthest from the root (i.e. deepest node in the tree), because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
```

```
    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
```

```
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
```

```
     to NULL indicating no more node waiting for suffix link
     reset.*/
    if (lastNewNode != NULL)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
      is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
      the edge being traversed and current character
      being processed is not  on the edge (we fall off
      the tree). In this case, we add a new internal node
      and a new leaf edge going out of that new node. This
      is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
      internal node created in last extensions of same
```

```
                phase which is still waiting for it's suffix link
                reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
           suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
```

```
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
              //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
```

```
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                                edgeLength(n->children[i]), maxHeight,
                                 substringStartIndex);
            }
        }
    }
    else if(n->suffixIndex > -1 &&
                (*maxHeight < labelHeight - edgeLength(n)))
    {
        *maxHeight = labelHeight - edgeLength(n);
        *substringStartIndex = n->suffixIndex;
    }
}

void getLongestRepeatedSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
//  printf("maxHeight %d, substringStartIndex %d\n", maxHeight,
//          substringStartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABABABA$");
```

```
    strcpy(text, "ABABABA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ATCGATCGA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "banana$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "abcpqrabpqpq$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "pqrpqpqabab$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAAAA$ is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
Longest Repeated Substring in ATCGATCGA$ is: ATCGA
Longest Repeated Substring in banana$ is: ana
Longest Repeated Substring in abcpqrabpqpq$ is: ab
Longest Repeated Substring in pqrpqpqabab$ is: ab
```

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1st lexicographically.

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that finding deepest node will take O(N).
So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text

2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

22. Suffix Tree Application 4 – Build Linear Time Suffix Array

Given a string, build it's Suffix Array
We have already discussed following two ways of building suffix array:

- Naive O(n^2Logn) algorithm
- Enhanced O(nLogn) algorithm

Please go through these to have the basic understanding.
Here we will see how to build suffix array in linear time using suffix tree.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

Lets consider string abcabxabcd.
It's suffix array would be:

```
0 6 3 1 7 4 2 8 9 5
```

Lets look at following figure:

This is suffix tree for String “abcabxabcd$”
If we do a DFS traversal, visiting edges in lexicographic order (we have been doing the same traversal in other Suffix Tree Application articles as well) and print suffix indices on leaves, we will get following:

```
10 0 6 3 1 7 4 2 8 9 5
```

“$” is lexicographically lesser than [a-zA-Z].
The suffix index 10 corresponds to edge with “$” label.
Except this 1st suffix index, the sequence of all other numbers gives the suffix array of the string.

So if we have a suffix tree of the string, then to get it’s suffix array, we just need to do a lexicographic order DFS traversal and store all the suffix indices in resultant suffix array, except the very 1st suffix index.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And and then create suffix array in linear time
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node
```

```
/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
```

```
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
```

```
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
            and move on to next phase*/
            break;
        }

        /*We will be here when activePoint is in middle of
          the edge being traversed and current character
          being processed is not  on the edge (we fall off
          the tree). In this case, we add a new internal node
          and a new leaf edge going out of that new node. This
          is Extension Rule 2, where a new leaf edge and a new
        internal node get created*/
        splitEnd = (int*) malloc(sizeof(int));
        *splitEnd = next->start + activeLength - 1;

        //New internal node
        Node *split = newNode(next->start, splitEnd);
        activeNode->children[text[activeEdge]] = split;

        //New leaf coming out of new internal node
        split->children[text[pos]] = newNode(pos, &leafEnd);
        next->start += activeLength;
        split->children[text[next->start]] = next;

        /*We got a new internal node here. If there is any
          internal node created in last extensions of same
          phase which is still waiting for it's suffix link
          reset, do it now.*/
        if (lastNewNode != NULL)
        {
        /*suffixLink of lastNewNode points to current newly
          created internal node*/
            lastNewNode->suffixLink = split;
        }

        /*Make the current newly created internal node waiting
          for it's suffix link reset (which is pointing to root
          at present). If we come across any other internal node
          (existing or newly created) in next extension of same
          phase, when a new leaf edge gets added (i.e. when
          Extension Rule 2 applies is any of the next extension
          of same phase) at that point, suffixLink of this node
          will point to that internal node.*/
        lastNewNode = split;
    }
```

```
        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
```

```
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}
```

```
void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
        suffixArray[(*idx)++] = n->suffixIndex;
    }
}
```

```
void buildSuffixArray(int suffixArray[])
{
    int i = 0;
    for(i=0; i< size; i++)
        suffixArray[i] = -1;
    int idx = 0;
    doTraversal(root, suffixArray, &idx);
    printf("Suffix Array for String ");
    for(i=0; i<size; i++)
        printf("%c", text[i]);
    printf(" is: ");
    for(i=0; i<size; i++)
        printf("%d ", suffixArray[i]);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "AAAAAAAAAA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
```

```
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "abcabxabcd$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "CCAAACCCGATTA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    return 0;
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2
Suffix Array for String GEEKSFORGEEKS is: 9 1 10 2 5 8 0 11 3 6 7 12 4
Suffix Array for String AAAAAAAAAA is: 9 8 7 6 5 4 3 2 1 0
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1
Suffix Array for String abcabxabcd is: 0 6 3 1 7 4 2 8 9 5
Suffix Array for String CCAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal of tree take O(N) to build suffix array.
So overall, it's linear in time and space.
Can you see why traversal is O(N) ?? Because a suffix tree of string of length N will have at most N-1 internal nodes and N leaves. Traversal of these nodes can be done in O(N).

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

23. Generalized Suffix Tree 1

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for substring check, searching all patterns, longest repeated substring and built suffix array (All linear time operations).

There are lots of other problems where multiple strings are involved.
e.g. pattern searching in a text file or dictionary, spell checker, phone book, Autocomplete, Longest common substring problem, Longest palindromic substring and More.

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.
A suffix tree made of a set of strings is known as Generalized Suffix Tree.
We will discuss a simple way to build Generalized Suffix Tree here for **two strings only**.
Later, we will discuss another approach to build Generalized Suffix Tree for **two or more strings**.

Here we will use the suffix tree implementation for one string discussed already and modify that a bit to build generalized suffix tree.

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y$ where # and $ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then
X#Y$ = xabxa#babxba$
If we run the code implemented at Ukkonen's Suffix Tree Construction – Part 6 for string xabxa#babxba$, we get following output:

(Click to see it clearly)

We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels #babxba$, a#babxba$ and bxa#babxba$,

we can remove babxba$ (belongs to 2nd input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:

(Click to see it clearly)

Below implementation is built on top of original implementation. Here we are removing unwanted characters on path labels. If a path label has "#" character in it, then we are trimming all characters after the "#" in that path label.

Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y$

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
```

```
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;
```

```
/*Increment remainingSuffixCount indicating that a
new suffix added to the list of suffixes yet to be
added in tree*/
remainingSuffixCount++;

/*set lastNewNode to NULL while starting a new phase,
 indicating there is no internal node waiting for
 it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children[text[activeEdge]] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
```

```
            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
            and move on to next phase*/
            break;
        }

        /*We will be here when activePoint is in middle of
          the edge being traversed and current character
          being processed is not  on the edge (we fall off
          the tree). In this case, we add a new internal node
          and a new leaf edge going out of that new node. This
          is Extension Rule 2, where a new leaf edge and a new
        internal node get created*/
        splitEnd = (int*) malloc(sizeof(int));
        *splitEnd = next->start + activeLength - 1;

        //New internal node
        Node *split = newNode(next->start, splitEnd);
        activeNode->children[text[activeEdge]] = split;

        //New leaf coming out of new internal node
        split->children[text[pos]] = newNode(pos, &leafEnd);
        next->start += activeLength;
        split->children[text[next->start]] = next;

        /*We got a new internal node here. If there is any
          internal node created in last extensions of same
          phase which is still waiting for it's suffix link
          reset, do it now.*/
        if (lastNewNode != NULL)
        {
        /*suffixLink of lastNewNode points to current newly
          created internal node*/
            lastNewNode->suffixLink = split;
        }

        /*Make the current newly created internal node waiting
          for it's suffix link reset (which is pointing to root
          at present). If we come across any other internal node
          (existing or newly created) in next extension of same
          phase, when a new leaf edge gets added (i.e. when
          Extension Rule 2 applies is any of the next extension
          of same phase) at that point, suffixLink of this node
          will point to that internal node.*/
        lastNewNode = split;
    }

    /* One suffix got added in tree, decrement the count of
      suffixes yet to be added.*/
    remainingSuffixCount--;
    if (activeNode == root && activeLength > 0) //APCFER2C1
    {
        activeLength--;
        activeEdge = pos - remainingSuffixCount + 1;
    }
    else if (activeNode != root) //APCFER2C2
    {
        activeNode = activeNode->suffixLink;
    }
}
```

```
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#') //Trim unwanted characters
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
```

```
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
//  strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    strcpy(text, "xabxa#babxba$"); buildSuffixTree();
    return 0;
}
```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```
# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
```

```
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]
```

If two strings are of size M and N, this implementation will take O(M+N) time and space. If input strings are not concatenated already, then it will take 2(M+N) space in total, M+N space to store the generalized suffix tree and another M+N space to store concatenated string.

Followup:
Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.
We will discuss another approach to build generalized suffix tree soon where we will need only one unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

24. Suffix Tree Application 5 – Longest Common Substring

Given two strings X and Y, find the Longest Common Substring of X and Y.

Naive [O(N*M^2)] and Dynamic Programming [O(N*M)] approaches are already discussed here.

In this article, we will discuss a linear time approach to find LCS using suffix tree (The 5th Suffix Tree Application).
Here we will build generalized suffix tree for two strings X and Y as discussed already at: Generalized Suffix Tree 1

Lets take same example (X = xabxa, and Y = babxba) we saw in Generalized Suffix Tree 1.
We built following suffix tree for X and Y there:

(Click to see it clearly)

This is generalized suffix tree for xabxa#babxba$
In above, leaves with suffix indices in [0,4] are suffixes of string xabxa and leaves with suffix indices in [6,11] are suffixes of string babxa. Why ??
Because in concatenated string xabxa#babxba$, index of string xabxa is 0 and it's length is 5, so indices of it's suffixes would be 0, 1, 2, 3 and 4. Similarly index of string babxba is 6 and it's length is 6, so indices of it's suffixes would be 6, 7, 8, 9, 10 and 11.

With this, we can see that in the generalized suffix tree figure above, there are some internal nodes having leaves below it from

- both strings X and Y (i.e. there is at least one leaf with suffix index in [0,4] and one leaf with suffix index in [6, 11]
- string X only (i.e. all leaf nodes have suffix indices in [0,4])
- string Y only (i.e. all leaf nodes have suffix indices in [6,11])

Following figure shows the internal nodes marked as "XY", "X" or "Y" depending on which string the leaves belong to, that they have below themselves.

(Click to see it clearly)

What these "XY", "X" or "Y" marking mean ?
Path label from root to an internal node gives a substring of X or Y or both.
For node marked as XY, substring from root to that node belongs to both strings X and Y.
For node marked as X, substring from root to that node belongs to string X only.
For node marked as Y, substring from root to that node belongs to string Y only.

By looking at above figure, can you see how to get LCS of X and Y ?
By now, it should be clear that how to get common substring of X and Y at least.
If we traverse the path from root to nodes marked as XY, we will get common substring of X and Y.

Now we need to find the longest one among all those common substrings.
Can you think how to get LCS now ? Recall how did we get Longest Repeated Substring in a given string using suffix tree already.
The path label from root to the deepest node marked as XY will give the LCS of X and Y. The deepest node is highlighted in above figure and path label “abx” from root to that node is the LCS of X and Y.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for two strings
// And then we find longest common substring of the two input strings
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
```

```
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;
```

```
/*set lastNewNode to NULL while starting a new phase,
 indicating there is no internal node waiting for
 it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children[text[activeEdge]] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children[text[activeEdge]] =
                                    newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
```

```
                and move on to next phase*/
                break;
            }

            /*We will be here when activePoint is in middle of
              the edge being traversed and current character
              being processed is not  on the edge (we fall off
              the tree). In this case, we add a new internal node
              and a new leaf edge going out of that new node. This
              is Extension Rule 2, where a new leaf edge and a new
            internal node get created*/
            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;

            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
```

```
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        //print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
         //   if (leaf == 1 && n->start != -1)
           //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                    edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#')
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
       // printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
```

```
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}
```

```
int doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                ret = doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);

                if(n->suffixIndex == -1)
                    n->suffixIndex = ret;
                else if((n->suffixIndex == -2 && ret == -3) ||
                    (n->suffixIndex == -3 && ret == -2) ||
                    n->suffixIndex == -4)
                {
                    n->suffixIndex = -4;//Mark node as XY
                    //Keep track of deepest node
                    if(*maxHeight < labelHeight)
                    {
                        *maxHeight = labelHeight;
                        *substringStartIndex = *(n->end) -
```

```
                        labelHeight + 1;
                }
            }
        }
    }
    else if(n->suffixIndex > -1 && n->suffixIndex < size1)//suffix of X
        return -2;//Mark node as X
    else if(n->suffixIndex >= size1)//suffix of Y
        return -3;//Mark node as Y
    return n->suffixIndex;
}

void getLongestCommonSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No common substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 7;
    printf("Longest Common Substring in xabxac and abcabxabcd is: ");
    strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 10;
    printf("Longest Common Substring in xabxaabxa and babxba is: ");
    strcpy(text, "xabxaabxa#babxba$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 14;
    printf("Longest Common Substring in GeeksforGeeks and GeeksQuiz is
    strcpy(text, "GeeksforGeeks#GeeksQuiz$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 26;
    printf("Longest Common Substring in OldSite:GeeksforGeeks.org");
    printf(" and NewSite:GeeksQuiz.com is: ");
    strcpy(text, "OldSite:GeeksforGeeks.org#NewSite:GeeksQuiz.com$");
    buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
```

```
    size1 = 6;
    printf("Longest Common Substring in abcde and fghie is: ");
    strcpy(text, "abcde#fghie$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Common Substring in pqrst and uvwxyz is: ");
    strcpy(text, "pqrst#uvwxyz$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Common Substring in xabxac and abcabxabcd is: abxa, of length: 4
Longest Common Substring in xabxaabxa and babxba is: abx, of length: 3
Longest Common Substring in GeeksforGeeks and GeeksQuiz is: Geeks, of length: 5
Longest Common Substring in OldSite:GeeksforGeeks.org and
NewSite:GeeksQuiz.com is: Site:Geeks, of length: 10
Longest Common Substring in abcde and fghie is: e, of length: 1
Longest Common Substring in pqrst and uvwxyz is: No common substring
```

If two strings are of size M and N, then Generalized Suffix Tree construction takes O(M+N) and LCS finding is a DFS on tree which is again O(M+N).
So overall complexity is linear in time and space.

Followup:

1. Given a pattern, check if it is substring of X or Y or both. If it is a substring, find all it's occurrences along with which string (X or Y or both) it belongs to.
2. Extend the implementation to find LCS of more than two strings
3. Solve problem 1 for more than two strings
4. Given a string, find it's Longest Palindromic Substring

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Suffix Tree Application 6 – Longest Palindromic Substring
- Generalized Suffix Tree 1

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above