kasai’s Algorithm for Construction of LCP array from Suffix Array - GeeksforGeeks

Background

Suffix Array : A suffix array is a sorted array of all suffixes of a given string.

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
```

The suffix array for “banana” :

suffix[] = {5, 3, 1, 0, 4, 2}

Once Suffix array is built, we can use it to efficiently search a pattern in a text. For example, we can use Binary Search to find a pattern (Complete code for the same is discussed here)

LCP Array

The Binary Search based solution discussed here takes O(m*Logn) time where m is length of the pattern to be searched and n is length of the text. With the help of LCP array, we can search a pattern in O(m + Log n) time. For example, if our task is to search “ana” in “banana”, m = 3, n = 5.

LCP Array is an array of size n (like Suffix Array). A value lcp[i] indicates length of the longest common prefix of the suffixes inexed by suffix[i] and suffix[i+1]. suffix[n-1] is not defined as there is no suffix after it.

```
txt[0..n-1] = "banana"
suffix[]  = {5, 3, 1, 0, 4, 2|
lcp[]     = {1, 3, 0, 0, 2, 0}

Suffixes represented by suffix array in order are:
{"a", "ana", "anana", "banana", "na", "nana"}



lcp[0] = Longest Common Prefix of "a" and "ana"     = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
```

```
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

How to construct LCP array?

LCP array construction is done two ways:

1) Compute the LCP array as a byproduct to the suffix array (Manber & Myers Algorithm)

2) Use an already constructed suffix array in order to compute the LCP values. (Kasai Algorithm).

There exist algorithms that can construct Suffix Array in O(n) time and therefore we can always construct LCP array in O(n) time. But in the below implementation, a O(n Log n) algorithm is discussed.

kasai's Algorithm

In this article kasai's Algorithm is discussed. The algorithm constructs LCP array from suffix array and input text in O(n) time. The idea is based on below fact:

Let lcp of suffix beginning at txt[i[be k. If k is greater than 0, then lcp for suffix beginning at txt[i+1] will be at-least k-1. The reason is, relative order of characters remain same. If we delete the first character from both suffixes, we know that at least k characters will match. For example for substring "ana", lcp is 3, so for string "na" lcp will be at-least 2. Refer this for proof.

Below is C++ implementation of Kasai's algorithm.

```
// C++ program for building LCP array for given text
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;  // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
           (a.rank[0] < b.rank[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
vector<int> buildSuffixArray(string 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].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

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

    // At his point, all suffixes are sorted according to first
    // 2 characters.  Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n];  // This array is needed to get the index in suffixes[]
    // from original index.  This mapping is needed to get
    // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
        {
            // If first rank and next ranks are same as that of previous
            // suffix in array, assign the same new rank to this suffix
            if (suffixes[i].rank[0] == prev_rank &&
                    suffixes[i].rank[1] == suffixes[i-1].rank[1])
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = rank;
            }
            else // Otherwise increment rank and assign
```

```
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = ++rank;
            }
            ind[suffixes[i].index] = i;
        }

        // Assign next rank to every suffix
        for (int i = 0; i < n; i++)
        {
            int nextindex = suffixes[i].index + k/2;
            suffixes[i].rank[1] = (nextindex < n)?
                                  suffixes[ind[nextindex]].rank[0]: -1;
        }

        // Sort the suffixes according to first k characters
        sort(suffixes, suffixes+n, cmp);
    }

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

    // Return the suffix array
    return  suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
    // invSuff[5] would store 0.  This is used to get next
    // suffix string from suffix array.
    vector<int> invSuff(n, 0);

    // Fill values in invSuff[]
    for (int i=0; i < n; i++)
```

```
        invSuff[suffixArr[i]] = i;

    // Initialize length of previous LCP
    int k = 0;

    // Process all suffixes one by one starting from
    // first suffix in txt[]
    for (int i=0; i<n; i++)
    {
        /* If the current suffix is at n-1, then we don’t
           have next substring to consider. So lcp is not
           defined for this substring, we put zero. */
        if (invSuff[i] == n-1)
        {
            k = 0;
            continue;
        }

        /* j contains index of the next substring to
           be considered  to compare with the present
           substring, i.e., next string in suffix array */
        int j = suffixArr[invSuff[i]+1];

        // Directly start matching from k'th index as
        // at-least k-1 characters will match
        while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
            k++;

        lcp[invSuff[i]] = k; // lcp for the present suffix.

        // Deleting the starting character from the string.
        if (k>0)
            k--;
    }

    // return the constructed lcp array
    return lcp;
}

// Utility function to print an array
void printArr(vector<int>arr, int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
```

```
    cout << endl;
}

// Driver program
int main()
{
    string str = "banana";

    vector<int>suffixArr = buildSuffixArray(str, str.length());
    int n = suffixArr.size();

    cout << "Suffix Array : \n";
    printArr(suffixArr, n);

    vector<int>lcp = kasai(str, suffixArr);

    cout << "\nLCP Array : \n";
    printArr(lcp, n);
    return 0;
}
```

Output:

```
Suffix Array :
5 3 1 0 4 2

LCP Array :
1 3 0 0 2 0
```

Illustration:

```
txt[]     = "banana",  suffix[]  = {5, 3, 1, 0, 4, 2|

Suffix array represents
{"a", "ana", "anana", "banana", "na", "nana"}

Inverse Suffix Array would be
invSuff[] = {3, 2, 5, 1, 4, 0}
```

LCP values are evaluated in below order

We first compute LCP of first suffix in text which is “**banana**“. We need next suffx in suffix array to compute LCP (Remember lcp[i] is defined as Longest Common Prefix of suffix[i] and suffix[i+1]). **To find the next suffix in suffixArr[], we use SuffInv[]**. The next suffix is “na”. Since there is no common prefix between “banana” and “na”, the value of LCP for “banana” is 0 and it is at index 3 in suffix array, so we fill **lcp[3]** as 0.

Next we compute LCP of second suffix which “**anana**“. Next suffix of “anana” in suffix array is “banana”. Since there is no common prefix, the value of LCP for “anana” is 0 and it is at index 2 in suffix array, so we fill **lcp[2]** as 0.

Next we compute LCP of third suffix which “**nana**“. Since there is no next suffix, the value of LCP for “nana” is not defined. We fill **lcp[5]** as 0.

Next suffix in text is “ana”. Next suffix of “**ana**” in suffix array is “anana”. Since there is a common prefix of length 3, the value of LCP for “ana” is 3. We fill **lcp[1]** as 3.

Now we lcp for next suffix in text which is “**na**“. This is where Kasai’s algorithm uses the trick that LCP value must be at least 2 because previous LCP value was 3. Since there is no character after “na”, final value of LCP is 2. We fill **lcp[4]** as 2.

Next suffix in text is “**a**“. LCP value must be at least 1 because previous value was 2. Since there is no character after “a”, final value of LCP is 1. We fill **lcp[0]** as 1.

We will soon be discussing implementation of search with the help of LCP array and how LCP array helps in reducing time complexity to O(m + Log n).

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