Suffix Array | Set 2 (nLogn Algorithm)

A suffix array is a sorted array [illegible] finition is similar to Suffix Tree which is compressed trie of all suffixes [illegible]

```
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" is {5, 3, 1, 0, 4, 2}
```

We have discussed **Naive algorithm** for construction of suffix array. The Naive algorithm is to consider all suffixes, sort them using a O(nLogn) sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is $O(n^2Logn)$ where n is the number of characters in the input string.

In this post, a **O(nLogn) algorithm** for suffix array construction is discussed. Let us first discuss a O(n * Logn * Logn) algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.

We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than 2n. The important point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in O(nLogn) time using a nLogn sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in O(1) time (we need to compare only two values, see the below example and code).

The sort function is called O(Logn) times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes O(nLognLogn). See http://www.stanford.edu/class/cs97si/suffix-array.pdf for more details.

Let us build suffix array the example string “banana” using above algorithm.

Sort according to first two characters Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do “str[i] – ‘a’” for ith suffix of strp[]

Index	Suffix	Rank
0	banana	1
1	anana	0
2	nana	13
3	ana	0
4	na	13
5	a	0

For every character, we also store rank of next adjacent character, i.e., the rank of character at str[i + 1] (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

Index	Suffix	Rank	Next Rank
0	banana	1	0
1	anana	0	13
2	nana	13	0
3	ana	0	13
4	na	13	0
5	a	0	-1

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	0	13
3	ana	0	13
0	banana	1	0
2	nana	13	0
4	na	13	0

Sort according to first four character

Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0

as new rank to first suffix. For assigning ranks to remaining suffixes, we consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

```
Index       Suffix          Rank
  5          a               0     [Assign 0 to first]
  1          anana           1     (0, 13) is different from previous
  3          ana             1     (0, 13) is same as previous
  0          banana          2     (1, 0) is different from previous
  2          nana            3     (13, 0) is different from previous
  4          na              3     (13, 0) is same as previous
```

For every suffix str[i], also store rank of next suffix at str[i + 2]. If there is no next suffix at i + 2, we store next rank as -1

```
Index       Suffix          Rank        Next Rank
  5          a               0            -1
  1          anana           1             1
  3          ana             1             0
  0          banana          2             3
  2          nana            3             3
  4          na              3             -1
```

Sort all Suffixes according to rank and next rank.

```
Index       Suffix          Rank        Next Rank
  5          a               0            -1
  3          ana             1             0
  1          anana           1             1
  0          banana          2             3
  4          na              3            -1
  2          nana            3             3
```

C++

```cpp
// C++ program 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; // 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
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].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 this 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
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    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;
}
```

Java

```
// Java code for implementation of above approach
import java.util.*;
class GFG
{
    // Class to store information of a suffix
    public static class Suffix implements Comparable<Suffix>
    {
        int index;
        int rank;
        int next;

        public Suffix(int ind, int r, int nr)
        {
            index = ind;
            rank = r;
            next = nr;
        }

        // A comparison function used by sort()
        // to compare two suffixes.
        // Compares two pairs, returns 1
        // if first pair is smaller
        public int compareTo(Suffix s)
        {
            if (rank != s.rank) return Integer.compare(rank, s.rank);
            return Integer.compare(next, s.next);
        }
    }

    // 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
    public static int[] suffixArray(String s)
```

```
    {
        int n = s.length();
        Suffix[] su = new Suffix[n];

        // Store suffixes and their indexes in
        // an array of classes. The class is needed
        // to sort the suffixes alphabatically and
        // maintain their old indexes while sorting
        for (int i = 0; i < n; i++)
        {
            su[i] = new Suffix(i, s.charAt(i) - '$', 0);
        }
        for (int i = 0; i < n; i++)
            su[i].next = (i + 1 < n ? su[i + 1].rank : -1);

        // Sort the suffixes using the comparison function
        // defined above.
        Arrays.sort(su);

        // At this 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 = new int[n];

        // This array is needed to get the index in suffixes[]
        // from original index. This mapping is needed to get
        // next suffix.
        for (int length = 4; length < 2 * n; length <<= 1)
        {

            // Assigning rank and index values to first suffix
            int rank = 0, prev = su[0].rank;
            su[0].rank = rank;
            ind[su[0].index] = 0;
            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 (su[i].rank == prev &&
                    su[i].next == su[i - 1].next)
                {
                    prev = su[i].rank;
                    su[i].rank = rank;
                }
                else
                {
                    // Otherwise increment rank and assign
                    prev = su[i].rank;
                    su[i].rank = ++rank;
                }
                ind[su[i].index] = i;
            }

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

            // Sort the suffixes according
            // to first k characters
            Arrays.sort(su);
```

```
        }

        // Store indexes of all sorted
        // suffixes in the suffix array
        int[] suf = new int[n];

        for (int i = 0; i < n; i++)
            suf[i] = su[i].index;

        // Return the suffix array
        return suf;
    }

    static void printArr(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver Code
    public static void main(String[] args)
    {
        String txt = "banana";
        int n = txt.length();
        int[] suff_arr = suffixArray(txt);
        System.out.println("Following is suffix array for banana:");
        printArr(suff_arr, n);
    }
}

// This code is contributed by AmanKumarSingh
```

Output:

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

Note that the above algorithm uses standard sort function and therefore time complexity is O(nLognLogn). We can use Radix Sort here to reduce the time complexity to O(nLogn).

Please note that suffx arrays can be constructed in O(n) time also. We will soon be discussing O(n) algorithms.

References:

http://www.stanford.edu/class/cs97si/suffix-array.pdf

http://www.cbcb.umd.edu/confcour/Fall2012/lec14b.pdf

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

Recommended Posts:

kasai's Algorithm for Construction of LCP array from Suffix Array

Suffix Tree Application 4 - Build Linear Time Suffix Array

Boyer Moore Algorithm | Good Suffix heuristic

Suffix Array | Set 1 (Introduction)

Counting k-mers via Suffix Array

Count of distinct substrings of a string using Suffix Array

Find starting index for every occurence of given array B in array A using Z-Algorithm

Count number of increasing sub-sequences : O(NlogN)

Z algorithm (Linear time pattern searching Algorithm)

Generalized Suffix Tree 1

Longest prefix which is also suffix

Pattern Searching using Suffix Tree

Queries for number of distinct integers in Suffix

Ukkonen's Suffix Tree Construction - Part 1

Ukkonen's Suffix Tree Construction - Part 2

Improved By : Akash Kumar 31, AmanKumarSingh

Article Tags : Advanced Data Structure Pattern Searching Suffix-Array

Practice Tags : Pattern Searching

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.