A tutorial on Suffix Arrays

Hello @all,

129 This text will focus on the construction of Suffix Arrays, it will aim to explain what they are and what they are used for and hopefully some examples will be provided (it will be mainly simple applications so that the concepts don't get too attached to the theoretical explanation).

71 As usual, this follows my somewhat recent series of tutorials in order to make the reference post with links as complete as possible!

- **What is a Suffix Array?**

In simple terms, a **suffix array** is just a sorted array of all the suffixes of a given string.

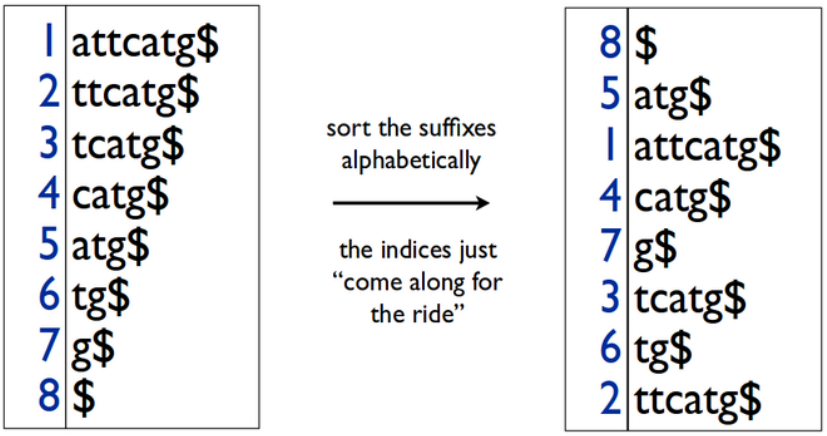
As a data structure, it is widely used in areas such as data compression, bioinformatics and, in general, in any area that deals with strings and string matching problems, so, as you can see, it is of great importance to know efficient algorithms to construct a suffix array for a given string.

Please note that on this context, the name suffix is the exact same thing as substring, as you can see from the wikipedia link provided.

A suffix array will contain integers that represent the starting indexes of the all the suffixes of a given string, **after** the aforementioned suffixes are sorted.

On some applications of suffix arrays, it is common to paddle the string with a special character (like #, @ or $) that is **not** present on the alphabet that is being used to represent the string and, as such, it's considered to be smaller than all the other characters. (The reason why these special characters are used will hopefully be clearer ahead in this text)

And, as a picture it's worth more than a thousand words, below is a small scheme which represents the several suffixes of a string (on the left) along with the suffix array for the same string (on the right). The original string is **attcatg$**.



The above picture describes what we want to do, and our goal with this text will be to explore different ways of doing this in the hope of obtaining a good solution.

We will enumerate some popular algorithms for this task and will actually implement some of them in C++ (as you will see, some of them are trivial to implement but can be too slow, while others have faster execution times at the cost of both implementation and memory complexity).

- **The naive algorithm**

We shall begin our exploration of this very interesting topic by first studying the most naive algorithm available to solve our problem, which is also the most simple one to implement.

The key idea of the naive algorithm is using a **good comparison-based sorting algorithm** to sort all the suffixes of a given string in the fastest possible way. **Quick-sort** does this task very well.

However we should remind ourselves that we are sorting strings, so, either we use the **overloaded < sign** to serve as a "comparator" for strings (this is done internally in C++ for the string data type) or we write our own string comparison function, which is basically the same thing regarding time complexity, with the former alternative consuming us more time on the writing of code. As such, on my own implementation I chose to keep things simple and used the built-in sort() function applied to a vector of strings. As to compare two strings, we are forced to iterate over all its characters, the time complexity to compare strings is O(N), which means that:

On the naive approach, we are sorting N strings with an O(N log N) comparison based sorting algorithm. As comparing strings takes O(N) time, we can conclude that the time complexity of our naive approach is $O(N^2 \log N)$

After sorting all the strings, we need to be able to "retrieve" the original index that each string had initially so we can actually build

the suffix array itself.

[Sidenote: As written on the image, the indexes just "come along for the ride".

To do this, I simply used a map as an auxiliary data structure, such that the keys are the strings that will map to the values which are the original indexes the strings had on the original array. Now, retrieving these values is trivial.]

Below, you can find the code for the naive algorithm for constructing the **Suffix Array** of a given string entered by the user as input:

```
//Naive algorithm for the construction of the suffix array of a given string
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    string s;
    map<string,int> m;
    cin >> s;
    vector<string> v;
    for(int i = 0; i < s.size();i++)
    {
        m[s.substr(i,s.size()-i)] = i;
        v.push_back(s.substr(i,s.size()-i));
    }
    sort(v.begin(),v.end());
    for(int i = 0; i < v.size();i++)
    {
        cout << m[v[i]] << endl;
    }
    return 0;
}
```

As you can see by the above code snippet, the implementation of the naive approach is pretty straightforward and very robust as little to virtually no space for errors is allowed if one uses built-in sorting functions.

However, such simplicity comes with an associated cost, and on this case, such cost is paid with a relatively high time complexity which is actually impractical for most problems. So, we need to tune up this approach a bit and attempt to devise a better algorithm.

This is what will be done on the next section.

- **A clever approach of building the Suffix Array of a given string**

As noted above, **Suffix Array construction** is **simple**, but **an efficient Suffix Array construction** is **hard**.

However, after some thinking we can actually have a very defined idea of why we are performing so badly on such construction.

The reason why we are doing badly on the construction of the SA is because we are **NOT EXPLOITING** the fact that the strings we are sorting, are actually all part of the **SAME** original string, and not random, unrelated strings.

However, how can this observation help us?

This observation can help us greatly because now we can actually use tuples that contain only some characters of the string (which we will group in powers of two) such that we can sort the strings in a more ordered fashion by their first two characters, then we can improve on and sort them by their first four characters and so on, until we have reached a length such that we can be sure all the strings are themselves sorted.

With this observation at hand, we can actually cut down the execution time of our SA construction algorithm from $O(N^2 \log N)$ to $O(N \log^2 N)$.

Using the amazing work done by @gamabunta, I can provide his explanation of this approach, along with his pseudo-code and later improve a little bit upon it by actually providing an actual C++ implementation of this idea:

@gamabunta's work

Let us consider the original array or suffixes, **sorted only according to the first 2 character**. If the first 2 character is the same, we consider that the strings have the same **sort index**.

```
Sort-Index Suffix-Index

    0       10: i
    1        7: ippi
    2        1: ississippi
    2        4: issippi
    3        0: mississippi
    4        9: pi
    5        8: ppi
    6        3: sissippi
    6        6: sippi
    7        2: ssissippi
    7        5: ssippi
```

Now, we wish to use the above array, and **sort the suffixes according to their first 4 characters**. To achieve this, we can assign **2-tuples** to each string. The first value in the **2-tuple** is the **sort-index** of the respective suffix, from above. The second value in the **2-**

tuple is the **sort-index** of the suffix that starts **2 positions later**, again from above.

If the length of the suffix is less than 2 characters, then we can keep the second value in the **2-tuple** as **-1**.

```
Sort-Index Suffix-Index               Suffix-Index
                                    after first 2 chars
                                   and 2-tuple assigned

    0       10: i               -1   (0, -1)
    1        7: ippi             9   (1, 4)
    2        1: ississippi       3   (2, 6)
    2        4: issippi          6   (2, 6)
    3        0: mississippi      2   (3, 7)
    4        9: pi              -1   (4, -1)
    5        8: ppi             10   (5, 0)
    6        3: sissippi         5   (6, 7)
    6        6: sippi            8   (6, 5)
    7        2: ssissippi        4   (7, 2)
    7        5: ssippi           7   (7, 1)
```

Now, we can call **quick-sort** and sort the suffixes according to their first 4 characters by using the **2-tuples** we constructed above! The result would be

```
Sort-Index Suffix-Index

    0       10: i
    1        7: ippi
    2        1: ississippi
    2        4: issippi
    3        0: mississippi
    4        9: pi
    5        8: ppi
    6        3: sissippi
    7        6: sippi
    8        2: ssissippi
    9        5: ssippi
```

Similarly constructing the **2-tuples** and performing **quick-sort** again will give us suffixes sorted by their **first 8 characters**.

Thus, we can sort the suffixes by the following pseudo-code

```
SortIndex[][] = { 0 }

for i = 0 to N-1
    SortIndex[0][i] = order index of the character at A[i]

doneTill = 1
step = 1

while doneTill < N
    L[] = { (0,0,0) } // Array of 3 tuples

    for i = 0 to N-1
        L[i] = ( SortIndex[step - 1][i],
                 SortIndex[step - 1][i + doneTill],
                 i
               )
    // We need to store the value of i to be able to retrieve the index

    sort L

    for i = 0 to N-1
        SortIndex[step][L[i].thirdValue] =
         SortIndex[step][L[i-1].thirdValue], if L[i] and L[i-1] have the same
first and second values
            i, otherwise

    ++step
    doneTill *= 2
```

The above algorithm will find the Suffix Array in **O(N $\log^2$ N)**.

end of @gamabunta's work

Below you can find a C++ implementation of the above pseudo-code:

```
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;
```

```
#define MAXN 65536
#define MAXLG 17

char A[MAXN];

struct entry
{
    int nr[2];
    int p;
} L[MAXN];

int P[MAXLG][MAXN];
int N,i;
int stp, cnt;

int cmp(struct entry a, struct entry b)
{
    return a.nr[0]==b.nr[0] ?(a.nr[1]<b.nr[1] ?1: 0): (a.nr[0]<b.nr[0] ?1: 0);
}

int main()
{
    gets(A);
    for(N=strlen(A), i = 0; i < N; i++)
        P[0][i] = A[i] - 'a';

    for(stp=1, cnt = 1; cnt < N; stp++, cnt *= 2)
    {
        for(i=0; i < N; i++)
        {
            L[i].nr[0]=P[stp- 1][i];
            L[i].nr[1]=i +cnt <N? P[stp -1][i+ cnt]:-1;
            L[i].p= i;
        }
        sort(L, L+N, cmp);
        for(i=0; i < N; i++)
            P[stp][L[i].p] =i> 0 && L[i].nr[0]==L[i-1].nr[0] && L[i].nr[1] == L[i- 1].nr[1]
? P[stp][L[i-1].p] : i;
    }
    return 0;
}
```

This concludes the explanation of a more efficient approach on building the suffix array for a given string. The runtime is, as said above, **O(N $\log^2$ N)**.

- **Constructing (and explaining) the LCP array**

The LCP array (Longest Common Prefix) is an auxiliary data structure to the suffix array. It stores the lengths of the longest common prefixes between pairs of consecutive suffixes in the suffix array.

So, if one has built the Suffix Array, it's relatively simple to actually build the LCP array.

In fact, using once again @gamabunta's amazing work, below there is the pseudo-code which allows one to efficiently find the LCP array:

We can use the **SortIndex** array we constructed above to find the **Longest Common Prefix**, between any two prefixes.

```
FindLCP (x, y)
    answer = 0

    for k = ceil(log N) to 0
        if SortIndex[k][x] = SortIndex[k][y]
            // sort-index is same if the first k characters are same
            answer += 2^k
            // now we wish to find the characters that are same in the remaining
strings
            x += 2^k
            y += 2^k
```

The **LCP Array** is the array of **Longest Common Prefixes** between the **i^{th}** suffix and the **$(i-1)^{th}$** suffix in the Suffix Array. The above algorithm needs to be called **N** times to build the **LCP Array** in a total of **O(N log N)** time.

- **Moving on from here**

This post was actually the first long post I wrote about a subject which I'm not familiar with, **AT ALL**. This is always a risk I am also taking, but I tried to adhere only to the sub-topics I considered I mastered relatively well myself (at least, in theory, as I still don't think I could implement this correctly on a live contest or even a pratice problem... But, as I said many times, I'm here to work as hard as I can to learn as much as I can!)

I hope that what I wrote is, at least, decent and I did it basically as a good way of gathering information which is very spread over many papers and websites online, so that when people read this post they will be able to grasp the ideas for the naive solution as well as for the improvement presented as a better solution.

There are many interesting linear algorithms to "attack" this problem, with one of the most famous being the Skew Algorithm, which is lovely described on the link I provide here.

Besides this, there are several other algorithms which are also linear that exploit the relationship between Suffix Trees and the Suffix Array and that use linear sorting algorithm like radix sort, but, which I sadly don't yet understand which makes me unable to discuss them here.

However, I hope this little text does its job by at least gathering some useful information on a single post :)

I am also learning as I write these texts and this has helped me a lot on my evolution as a coder and I hope I can keep contributing to give all my best to this incredible cmmunity :D

Best regards,

Bruno Oliveira

arrays construction suffix tutorial

edited **24 Oct '13, 15:52**

asked **17 Aug '13, 05:33**

kuruma
16.6k●72●143●208
accept rate: 8%

1 Nice tutorial for start in suffix arrays.

And I really recommend Skew Algorithm. It's just so awesome and magic, that it takes my breath away first time I saw it :)

michal27 (17 Aug '13, 05:59)

I am glad you liked it @michal27 :) I will definitely try to learn Skew's Algorithm in a not so distant future :)

kuruma (17 Aug '13, 14:42)

1 Awesome post. Keep it up. :)

h43k3r (17 Aug '13, 23:42)

2 Nice explanation. Another very clean explanation: http://stackoverflow.com/questions/17761704/suffix-array-algorithm

rajneesh2k10 (16 Dec '13, 22:27)

1 I was searching for it. Thank you.

arnab_das (11 Mar '14, 17:45)

showing 5 of 9 show all

25 Answers:

oldest newest **most voted**

1 2 3 **next »**

16

Really,easy to understand from this tutorial!!!:-)

link | award points

answered **23 Aug '14, 15:36**

codemaster1994
2.2k●7●20●18
accept rate: 0%

3

very well explained :)

link | award points

answered **20 Aug '13, 00:18**

akashverma_123
109●7
accept rate: 3%

1 Thank you very much sir! It's always great to see our work appreciated and obviously, @gamabunta also helped me a lot here :)

kuruma (20 Aug '13, 01:06)

3

Since answer to the above question could not come in comments, i am writing a separate answer :

1. L stores the actual index in L.p ; and two rank tuples in L.nr[2]

   for a string starting at position p, its first half rank is in L.nr[0] and second half is at L.nr1.

   Now it depends upon the stage of algorithm what is the size of each half. It doubles in each step.

2. Comparison code says that :

   you want me to compare two strings L1, L2 ;

   each starting at pos L1.p and L2.p ; rank of first half of L1 is in L1.nr[0] ; similarly for L2.nr[0]

   if first half of L1 and L2 is same -> then to find the answer compare second half. i.e. L1.nr1 and L2.nr1

   otherwise if they are not same, then we know the answer i.e. compare L1.nr[0] and L2.nr1

Read this : for only understand suffix array :

TopCoder Suffix Array

Then understand that matrix P is required for computing LCP.

link | award points

edited **13 Jan '14, 17:04**

answered **13 Jan '14, 17:03**

ashishnegi001
162●1●3●7
accept rate: 0%

Thanks a lot for the post.... Now I understand the suffix array implementation clearly.

1 I think in the c++ implementation it should be

```
for(stp=1, cnt = 1; cnt < N; stp++, cnt *= 2)
```

instead of

```
for(stp=1, cnt = 1; cnt/2 < N; stp++, cnt /= 2)
```

link | award points

answered **09 Sep '13, 02:11**
saikrishna173
165●7●9●16
accept rate: 9%

Yes, obviously, otherwise it wouldnt terminate properly :) I've seen this code snippet on a paper, but, I've fixed it now :D

kuruma (11 Sep '13, 19:13)

Hello,

1 Not accordingly to the notation that @gamabunta used.

For instance if the suffix is **si**ppi, with sort-index 6, than 6 is the first value in the tuple.

As **ppi** is the suffix that starts 2 positions later, we search for its sort-index which is 5.

So tuple stays (6,5). It's all the same for remaining suffixes :)

Bruno

link | award points

answered **02 Oct '13, 13:10**
kuruma
16.6k●72●143●208
accept rate: 8%

awesome (y)

1

link | award points

answered **15 May '14, 17:59**
ndatta
11●1
accept rate: 0%

very Good.. :)

1

link | award points edited **12 Aug '14, 18:13**

answered **12 Aug '14, 10:44**
warlock_ankur
26●4
accept rate: 0%

Now, we can call quick-sort and sort the suffixes according to their first 4 characters by using the 2-tuples we constructed above! The result would be

1

can anyone please explain the above line???qsort is called in which tuple.....

link | award points

answered **23 Aug '14, 10:00**
nil96
172●7●16●30
accept rate: 6%

how tuples r used for sorting please explain???

1

link | award points

answered **23 Aug '14, 10:03**
nil96
172●7●16●30
accept rate: 6%

1

Now, we can call quick-sort and sort the suffixes according to their first 4 characters by using the 2-tuples we constructed above! The result would be:
6 6: **sis**sippi
7 3: **sipp**i
8 5: **ssis**sippi
9 2: **ssip**pi

Can anyone explain me the result of above step. How are the suffixes sorted according to first 4 characters after calling quick sort. I mean, how is '**sissippi**' is having a lower sort-index then '**sippi**' and simmilarly how is '**ssissippi**' having a lower sort-index than '**ssippi**' ? I think the correct order should be:

6 6: **sipp**i
7 3: **sis**sippi
8 5: **ssip**pi
9 2: **ssis**sippi

link | award points

answered **09 Oct '14, 21:41**

vaibhavatul47
41●**2**●**4**
accept rate: 0%

1 2 3 **next »**

Your answer

[hide preview] community wiki

Type the text

Privacy & Terms

Post Your Answer