About Me

Name: Surayans Tiwari
University: Birla Institute of Technology and Science,Pilani
Major: B.E. Mechanical Engineering
Email: f2013777@pilani.bits-pilani.ac.in
IRC: serious
Github: SuryanshTiwari
Homepage:suryanshtiwari.cleverapps.io

Background work and Programming Skills

Educational background

I am a fifth year student of BITS Pilani, India. I'm pursuing a double major in B.E. Mechanical and Msc Chemistry. I use Sublime Text for development and vim for SSH sessions. I am proficient in C, C++ Python, javascript.I have done **Data Structures and Algorithms, Object Oriented Programming, Operating Systems** as my elective courses in my college.

Programming Interest and why I want to work for Boost C++ libraries?

I started my journey in C++ 3 years back when I started practicing problems on data structures and algorithms on various online judges SPOJ, codeforces, Codechef. Since then I have won several programming contest coding in C++ .

C++ easily lets me convert my ideas into code. I like it mainly because it is an compiled language which gives you freedom to do so much things fast. Prototyping anything in C++ is very easy and requires less man-work than any other programming language. I am really fond of the boost C++ library and I guess it's one of the best available in the community.

I want to make it more better by contributing to its code and implementing some of the algorithms that I really think would be a great addition to Boost universe.

What is your interest in the project you are proposing?

Data structures and algorithms are used everywhere in todays software and my project will lead to using them more easily. I have researched a lot about the project and I think it will not only make Boost more handy but also will be a great addition for people who face severe difficulty in implementing big data structures and complex algorithms.

I am not only writing in just a perspective of an algorithms enthusiast and competitive programmer but also people who generally use data structures and algorithms in their software.

Rating

C++ 98/03 (traditional C++)	3
C++ 11/14 (modern C++)	4
C++ Standard Library	4
Boost C++ Libraries	4
Git	4

Project Proposal

I will be focussing mainly on integrating Boost c++ libraries with new algorithms and data Structures that are not yet implemented.

Here is a list -

- **String Hashing**

Hashing algorithms are helpful in solving a lot of problems. But they have a big flaw that sometimes they are not 100% deterministically correct because when there are plenty of strings, hashes may collide. However, in a wide majority of tasks this can be safely ignored as the probability of the hashes of two different strings colliding is still very small.he best and most widely used way to define the hash of a string Sis the following function:

hash(S)=S[0]+S[1]·P+S[2]·P^2+S[3]·P^3+...+S[N]·P^N where P is some number.

It is reasonable to make P a prime number roughly equal to the number of characters in the input alphabet. For example, if the input is composed of only lowercase letters of English alphabet,P=31 is a good choice. If the input may contain both uppercase and lowercase letters, then P=53 is a possible choice.

So in our hash function the user will first pass a string to hash and a MOD value such that the string hash will be computed modulo MOD , and then two arguments , the arguments will be the start and end index of the substring they want the hash for , we will then be using **modular inverse** to return the hash for the substring.

- **Fenwick Tree**

A Fenwick tree orbinary indexed tree is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers.

Let, f be some reversible function and A be an array of integers of length N.

Fenwick tree is a data structure which:

•calculates the value of functionf in the given range[l;r](i.e.f(Al,Al+1,…,Ar)) inO(lgn)time;

•updates the value of an element ofA in O(lgn)time;
•requires O(N) memory, or in other words, exactly the same memory required forA;
•is easy to use and code, especially, in the case of multidimensional arrays.

Fenwick tree is also called Binary Indexed Tree.

The most common application of Fenwick tree iscalculating the sum of a range(i.e.f(A1,A2,…,Ak)=A1+A2+..+Ak).

- **Suffix Array**

What is a suffix?
Let S be a string of length N. The ith suffix of S is substring S[i…n−1],i=0…n−1.

What is a suffix array?

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.

A Suffix Array will contain integers that represent thestarting indexes of the all the suffixes of a given string, after the aforementioned suffixes are sorted.

Example

LetS=abaab
All suffixes are as follows
0.abaab
1.baab
2.aab
3.ab
4.b
After sorting these strings:
2.aab
3.ab
0.abaab
4.b
1.baaab
Suffix Array for S will be (2,3,0,4,1).

Contruction of suffix array(O(nlog^2n) approach)

We can reduce comparison of two strings from O(N) to O(1) using the fact that, given strings are not random strings they are part of single string.Each string has something in common with others.

Let's see how we can use this fact. Let's sort the suffixes on basis of their first character and assign them rank.
If two are equal rank for them will be same.

0.a|baab
0.a|ab
0.a|b
1.b|aab
1.b|
Note:'|' denotes sorting is based on left part of '|'.

Now double the characters to take from each for sorting i.e. 2.
When we take string of two chars we can have two parts first containing 1 char other containing 1.
Let's compareabaabwithbaab, based on the first part, that of 1 character we can say thatabaabwill be always ranked abovebaabso skip further comparison.
Now compareabaabwithaabbased on first part both have same rank. Now we will compare their second half part,
second half part ofabaabis onlyband foraabbe isafor these we already know their ranks forb(i.e. wholebaab) is 1 anda(i.e. wholeab) is 0.
Hence abaab will be ranked above baab.
For the strings not having second part we will rank their second part highest i.e. -1 for examplebis not having2ndchar so its rank tuple will be (1, -1).
0.aa|b
1.ab|aab
1.ab|
2.b|
2.ba|ab
Now, in the next iteration, we sort 4-characters strings. This involves a lot of comparisons between different 4-characters strings.
How do we compare two 4-characters strings? Well, we could compare them character by character. That would be up to 4 operations per comparison.
But instead, we compare them by looking up the ranks of the two characters contained in them, using the rank table generated in the previous steps.
That rank represents the lexicographic rank from the previous 2-charater sort, so if any given 4-character string has a higher rank than another 4-character string, then it must be lexicographically greater somewhere in the first two characters.
Hence, if for two 4-characters string the rank is identical, they must be identical in the first two characters.
In other words, two look-ups in the rank table are sufficient to compare all 4 characters of the two 4-characters strings.

Similarly we can compare 8, 16, 32…strings in at most two integer comparisons.
i.e.O(1)comparison.

- **Suffix Tree**

In computer science, a suffix tree(also called position tree) is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix trees allow particularly fast implementations of many important string operations.

This algorithm builds a suffix tree for a given strings of length in O(nlog(k))) time, where k is the size of the alphabet (if k is considered to be a constant, the asymptotic behavior is linear).
The input to the algorithm are the stringsand its lengthn, which are passed as global variables.
The main function **build_tree** builds a suffix tree. It is stored as an array of structures**node**, where **node[0]** is the root of the tree.

In order to simplify the code, the edges are stored in the same structures: for each vertex its structure **node** stores the information about the edge between it and its parent. Overall each **node** stores the following information:

- **(l, r)**- left and right boundaries of the substring**s[l..r-1]**which correspond to the edge to this node,
- **par**- the parent node,
- **link**- the suffix link,
- **next**- the list of edges going out from this node.

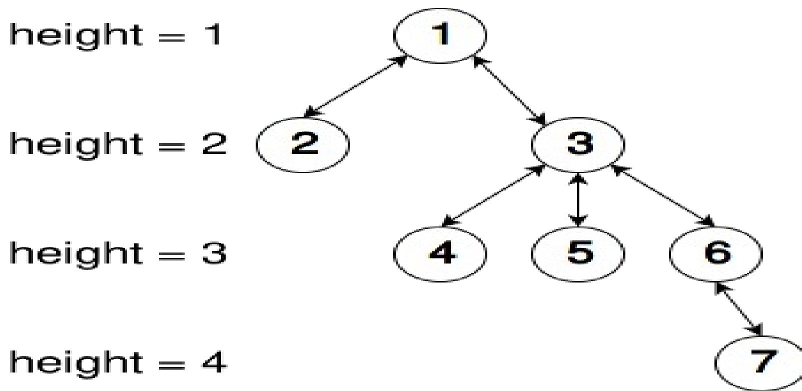
- **Lowest Common Ancestor**

In graph theory and computer science, the lowest common ancestor (LCA) of two nodes v and w in a tree or directed acyclic graph (DAG) T is the lowest (i.e. deepest) node that has both v and w as descendants, where we define each node to be a descendant of itself (so if v has a direct connection from w, w is the lowest common ancestor).

Given a tree G. Given queries of the form (v1,v2), for each query you need to find the lowest common ancestor (or least common ancestor), i.e. a vertex v that lies on the path from the root to v1 and the path from the root to v2, and the vertex should be the lowest. In other words, the desired vertex v is the most bottom ancestor of v1 and v2. It is obvious that their lowest common ancestor lies on a shortest path from v1 and v2. Also, if v1 is the ancestor of v2, v1 is their lowest common ancestor.
Before answering the queries, we need to do preprocessing. Run DFS from the root using preorder traversal and it will build an array list which stores the visit order of the vertices (current vertex is added to the list at the entrance to the vertex and after return from its child (children)). This may also be called an euler tour of the tree. It is clear that the size of this list will be O(N). We also need to build an array first[1...N] which stores, for each vertex i, its first occurance in list. That is, the first position in list such that list[first[i]]=i. Also by using the DFS we can find the height of each node (distance from root to it) and store it at height[1...N].

So how to answer the queries? Suppose the query is a pair of v1 and v2. Consider the elements in list between indices first[v1]and first[v2]. It is easy to notice that in this range there are LCA(v1,v2)

and many other peaks. However the LCA(v1,v2) can be uniquely determined, that is, the vertex with the lowest height.
Let's illustrate this idea. Consider the following graph:



In this example:

list height =={1,2,1,3,4,3,5,3,6,7,6,3,1}{1,2,1,2,3,2,3,2,3,4,3,2,1}

Thus, to answer the query, we just need to find the vertex with smallest height in the array list in the range from first[v1] to first[v2]. Thus, the objective of finding LCA is reduced to the RMQ problem (minimum in an interval problem).
If you use the sqrt-decomposition, it is possible to obtain a solution, answering each query in O(N−−√) with preprocessing in O(N)time.
If you use segment tree, you can answer each query in O(log N) with preprocessing in O(N) time.

- **Euler Totient Function**

E**uler's totient function, also known as phi-function ϕ(n), is the number of integers between 1 and n, inclusive, which are coprime to n.** Two numbers are coprime if their greatest common divisor equals 1 (1 is considered to be coprime to any number).

Here are values of ϕ(n) for the first few positive integers:

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
Phi(N)	**1**	**1**	**2**	**2**	**4**	**2**	**6**	**4**	**6**	**4**	**10**	**4**	**12**	**6**	**8**	**8**	**16**	**6**	**18**	**8**	**12**

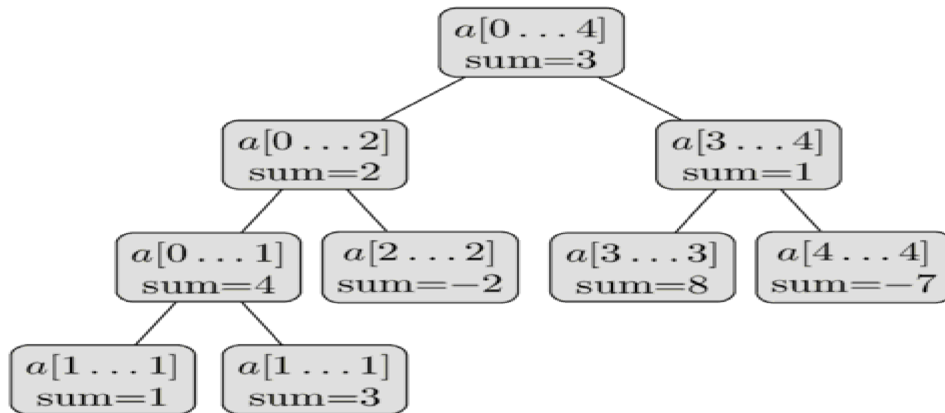
- **Segment Tree**

A Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. This includes finding the sum of consecutive array elements $a[l \dots r]$, or finding the minimum element in a such a range in $O(\log n)$ time. Between answering such queries the Segment Tree allows modifying the array by replacing one element, or even change the elements of a whole subsegment (e.g. assigning all elements $a[l \dots r]$ to any value, or adding a value to all element in the subsegment).

In general a Segment Tree is a very flexible data structure, and a huge number of problems can be solved with it. Additionally it is also possible to apply more complex operations and answer more complex queries . In particular the Segment Tree can be easily generalized to larger dimensions. For instance with a two-dimensional Segment Tree you can answer sum or minimum queries over some subrectangle of a given matrix. However only in $O(\log 2n)$ time.
One important property of Segment Trees is, that they require only a linear amount of memory. The standard Segment Tree requires $4n$vertices for working on an array of size n.

So, what is a Segment Tree?

We compute and store the sum of the elements of the whole array, i.e. the sum of the segment a[0…n−1]. We then split the array into two halves a[0…n/2] and a[n/2+1…n−1] and compute the sum of each halve and store them. Each of these two halves in turn also split in half, their sums are computed and stored. And this process repeats until all segments reach size 1. In other words we start with the segment a[0…n−1], split the current segment in half (if it has not yet become a segment containing a single element), and then calling the same procedure for both halves. For each such segment we store the sum of the numbers on it.
We can say, that these segments form a binary tree: the root of this tree is the segment a[0…n−1], and each vertex (except leaf vertices) has exactly two child vertices. This is why the data structure is called "Segment Tree", even though in most implementations the tree is not constructed explicitly
Here is a visual representation of such a Segment Tree over the array a=[1,3,−2,8,−7]:



- **Z- Function**

Suppose we are given a string s of length n. The Z-function for this string is an array of length n where the i-th element is equal to the greatest number of characters starting from the position i that coincide with the first characters of s.

In other words, z[i] is the length of the longest common prefix between s and the suffix of s starting at i.
Note. In this article, to avoid ambiguity, we assume 0-based indexes; that is: the first character of s has index 0 and the last one has index n−1.
The first element of Z-function, z[0], is generally not well defined. In this article we will assume it is zero (although it doesn't change anything in the algorithm implementation).

Applications

Search the substring
Number of distinct substrings in a string
String compression

- **Programming Competency**

I have written around 5000 lines of code in C++ and won several programming contest. Here is a link to my profile -

https://a2oj.com/profile?Username=terminated

https://www.hackerrank.com/terminated?hr_r=1

http://www.spoj.com/users/terminated/

https://github.com/SuryanshTiwari/directi

I have also won Hp think a thon(2015, 2016) two times and my team was shortlisted for ACM International Collegiate Programming Contest (ICPC) 2015.