Persistent segment trees – Explained with spoj problems

July 13, 2014 | anudeep2011 ⊕ Go to comments ∇ Leave a comment(19)

In this post I will introduce the concept of persistent data structures. We shall deal with a couple of problems : COT, MKTHNUM

Consider the following question :

Given a linked list, you need to support 2 operations. 1) Update the first x values. 2) Print the linked list after k'th update (k <= number of update operations made till now)

Simple solution is to create a new linked list after each update operation. print it when needed. Let us optimize it for memory.

We are only updating first x elements of the linked list. That means the new linked list will be almost same as previous linked list if x is small. We shall take advantage of this and avoid allocating memory for some elements. We can allocate memory for x elements, store new values in them, then point the next pointer of x'th node of new linked list to (x+1)'th node of previous linked list. This way we allocated only x units of memory per update operation.

Consider the following example.

Initial linked list : 1 2 3 4 5 6 7 8 9 10.
Update 1 : x = 4, new elements : 12 13 14 15
Update 2 : x = 2, new elements : 20 21
Update 3 : x = 6, new elements : 1 2 3 4 5 6
Update 4 : x = 1, new elements : 100

Initial linked list will look like this

```
head[0]-->[01]-->[02]-->[03]-->[04]-->[05]-->[06]-->[07]-->[08]-->[09]-->[10]
```

After first update total data will look like this

```
head[0]-->[01]-->[02]-->[03]-->[04]-->[05]-->[06]-->[07]-->[08]-->[09]-->[10]
                                      /
head[1]-->[12]-->[13]-->[14]-->[15]--'
```

We allocated memory for first 4 nodes, stored values in them, then next pointer of 4th node is pointing to 5th node of 1st linked list.
Now if we want to print first linked list, we start with head[0]. If we want to print 2nd linked list we start with head[1].
Data will look like this after all updates

```
head[0]-->[01]-->[02]-->[03]-->[04]-->[05]-->[06]-->[07]-->[08]-->[09]-->[10]
                                       /               |
head[1]-->[12]-->[13]-->[14]-->[15]---'                |
                        /                             /
head[2]-->[20]-->[21]--'                             /
                                                    /
head[3]-->[01]-->[02]-->[03]-->[04]-->[05]-->[06]--'
                  /
head[4]-->[100]--'
```

We are done with it. Memory used is O(n+x1+x2..+xm), note that also the time complexity is reduced from O(n*n) to O(n+x1+x2+..+xm). This is useful when sum of x is in O(n), in that case total complexity will be O(n) (memory and time).

What we have build above is a Persistent data structure. I am done with introducing you to Persistent data structures, now I will explain Persistent segment trees with this SPOJ problem : COT

10628. Count on a tree – SPOJ – COT – Editorial and solution implementation

You are given a tree with N nodes.The tree nodes are numbered from 1 to N.Each node has an integer weight.
We will ask you to perform the following operation:

u v k : ask for the **k**th minimum weight on the path from node **u** to node **v**

Note : Time limits are very strict. My O(N * log N) solution passed only with fast IO.
Our goal is to come up with O(N * log N) solution.

We can binary search for the answer. Now for a fixed value **x**, we need to answer the number of nodes with value less than **x** on the path from node **u** to node **v**. Let the number of such nodes be **y**. If **y** is less than **k**, we have to consider values greater than **x**. In other case consider values less than **x**.

Now the question is, given **u v x**, we need to answer the number of nodes with value less than **x** on the path from **u** to **v**.
Let f(**u**, **x**) be the number of nodes less than **x** on the path from **root** to **u**.
We can formulate the required answer as follows :
f(**u**, **x**) + f(**v**, **x**) – f(lca(**u**, **v**), **x**) – f(parent(lca(**u**, **v**)), **x**)
where lca(**u**, **v**) is Lowest common ancestor of **u** and **v**. It can be calculated in O(log N) (here and here)

Now the problem is further reduced, we just need to calculate function f. If the function f works in O(log N) time, then we have an O(log N * log N) solution per query. We shall see how function f can work in O(log N)

Let us use coordinate compression on values (here). Then we have all the values in the range [0, N).
Now assume that for each node of the tree, we have a segment tree build and ready to be used. segment tree is build in the following way : For a node **u**, all the nodes from **root** to **u** are considered, their respective compressed values are used to build the segment tree such that it will answer number of elements less than some **x** in O(log N). This is exactly what the function f has to calculate. Hence if we have N segment trees, one per node, we can solve the problem in O(log N * log N) per query. How to build such segment tree is left out. It is a standard problem.

Now the only issue left in this solution is building N segment trees, which may take O(N * N) space and time. Let us use the concept which I introduced above to reduce the memory. Note that the segment tree for a node **u** is made with compressed values of nodes from **root** to **u**. Now for any child **c** of **u**, its segment tree of **c** is build with same values of node **u**, except that one new value is added, that is the value of current node **c**. Consider the addition of this new value like an update operation on segment tree of **u**. Then the update operation will change O(log N) nodes from last level to the 1st level. There you go, the new segment tree of node **c** is almost same as segment tree of **u**, except for O(log N) nodes. Hence we can only allocate memory for this new O(log N) nodes and reuse other nodes from old segment tree. We are allocating memory for O(log N) nodes per each node in the tree. Hence the total memory used will be O(N * log N). Note that as the memory is reduced from O(N * N) to O(N * log N), time complexity is also reduced to O(N * log N).

We are done with the O(N * log N + M * log N * log N) solution. We construct the segment trees in O(N * log N), then answer each query in O(log N * log N) (binary search + segment tree query). As I told you the time limits for the problem are very strict, and this solution will not pass. We need to modify it to O((N+M) * log N) to get it accepted.

Do we really need binary search? Can we merge it with the segment tree query? Yes. Here is

how.
Start with the segment trees at four nodes (see the formula we derived to understand which four nodes). Now check the number of elements in left sub tree, if it is greater than **k**, then our answer is present in left sub tree. If not it is present in right sub tree. This way we can travel the 4 segment tress at once and get the answer in O(log N). Don't worry if you did not understand this last step, looking at the code will make it clear.

Click here for code
(You will get TLE if you submit it on SPOJ, adding fast input output will give AC, I removed that module to keep the code small and clean)

3946. K-th Number - MKTHNUM - Editorial and solution implementation

Given an array a[1 ... N] of different integer numbers, your program must answer a series of questions Q(i, j, k) in the form: "What would be the k-th number in a[i ... j] segment, if this segment was sorted?

There are solutions which work in O((N + M) * log N * log N) or O((N + M * log N) * log N * log N), those solutions will give AC. I will describe O((N+M) * log N) solution.

Let us assume we have a segment tree for all N * N ranges. That is for each i, j such that 1 <= i <= j <= N we have a segment tree ready to be used. In this case we can answer the query in O(log N) which is what we need to do. But building those segment trees will take O(N*N*N) time and memory. Like in previous problem let me concentrate on reducing memory which will in turn reduce time complexity.

Trick 1 : In segment tree for range (i, j) each node can be calculated from respective nodes in segm[illegible] tree [illegible] nge (1, i-1) [illegible] range [illegible] for range [illegible] segment [illegible] cause all input values are distinct, see why it wont work in other case)

Trick 2 : Segment tree for prefix i is almost same as segment tree for prefix i-1, except some O(log N) nodes that will change. So once again we can reduce the memory to O(N * log N).

Great, we are done with it. From O(N*N*N) memory we reduced to O(N * log N). Time complexity will be O(N * log N). There you go, O((N+M) * log N) solution.

Click here for code

Related Problems

Codechef – QUERY - http://www.codechef.com/problems/QUERY - Editorial

Codechef – SORTING - http://www.codechef.com/problems/SORTING – Editorial

Codeforces - http://codeforces.com/problemset/problem/226/E - Editorial

Comments were not working due to a bug. It is fine now. Also the contact me form was not working and so I lost about 30-40 messages i received in last 2-3 months. Sorry for the inconvenience. You can now click on contact me and mail me.
Do comment problems related to persistent segment trees, I will add them.
Also suggest me a light weight, clean and simple theme.
Share this post. Learn and let learn! 😀

Filed underData Structures, Segment trees, SPOJ| Tags:| | 17,381 views

« Heavy Light Decomposition When 2 guys talk, its not always about girls/sports ;) »

19 Comments.

Leave a comment ?

shaheen13 February 3, 2015 at 3:22 PM

For problem "COT",
you don't need to use faster IO.
just avoid "map".
instead of this , use binary search for data compression .
i just edited your , and got AC.
http://paste.ubuntu.com/10031841/

Reply

Aditya December 22, 2014 at 9:27 AM

A very powerful and detailed explanation. Thanks so much! Are there any other problems on SPOJ that require persistent segment trees so that I can practice the logic myself?

Reply

Rahul Kumar December 18, 2014 at 3:00 PM

i has basic understanding of segment tree.my doubt is,
we have to take array size of 2^n for make segment tree of n element array,then how to make segment tree of 30 or greater elements,because 2^30 = 1073741824
then how to take array of this larger size for make segment tree,how to implement ?

Reply

anudeep2011 December 26, 2014 at 9:32 PM