Merge Small to Large

Borworntat Dendumrongkul

IOI Thailand Task Team

Table of Contents

1 Merging Data Structure

2 Sack Technique

3 Lomsat gelral

Merging Data Structure

Sometimes, we want to merge 2 data structures together

Problem

Distinct Colors You are given a rooted tree consisting of n nodes. The nodes are numbered $1, 2, \ldots, n$ and node 1 is the root. Each node has a color.
For each node, determine the number of distinct colors present in its subtree.

Merging Data Structure

Naive Solution: we can DFS and count distinct color of each subtree with sets.

```
void dfs(int u, int p) {
    for(auto v: adj[u]) {
        if(v == p) {
            continue;
        }
        dfs(v, u);
        merge(u, v);
    }
}
```

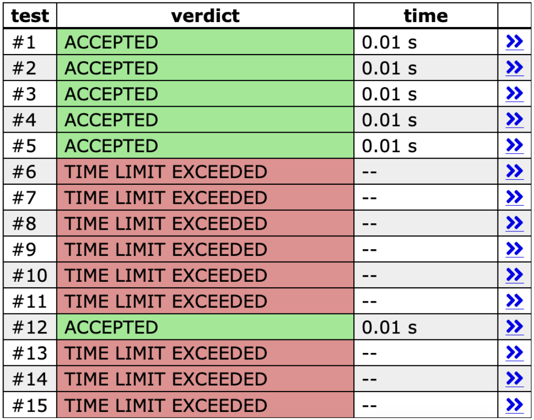
Merging Data Structure

By merging, we can easily do something like:

```
set<int> color[N];
void merge(int u, int v) {
    for(auto e: color[v]) {
        color[u].emplace(e);
    }
}
```

The function above has a time complexity $\mathcal{O}(n \log(n))$.

Merging Data Structure



test	verdict	time	
#1	ACCEPTED	0.01 s	»
#2	ACCEPTED	0.01 s	»
#3	ACCEPTED	0.01 s	»
#4	ACCEPTED	0.01 s	»
#5	ACCEPTED	0.01 s	»
#6	TIME LIMIT EXCEEDED	--	»
#7	TIME LIMIT EXCEEDED	--	»
#8	TIME LIMIT EXCEEDED	--	»
#9	TIME LIMIT EXCEEDED	--	»
#10	TIME LIMIT EXCEEDED	--	»
#11	TIME LIMIT EXCEEDED	--	»
#12	ACCEPTED	0.01 s	»
#13	TIME LIMIT EXCEEDED	--	»
#14	TIME LIMIT EXCEEDED	--	»
#15	TIME LIMIT EXCEEDED	--	»

As you can see, it's TLE because the running time of the algorithm is $\mathcal{O}(n^2 \log(n))$

Merging Data Structure

We have to use the idea of “merging small to large”. So, we can modify the *merge* function to

```
void merge(set<int> &a, set<int> &b) {
    if(a.size() < b.size()) {
        swap(a, b);
    }
    for(auto e: b) {
        a.emplace(e);
    }
}
```

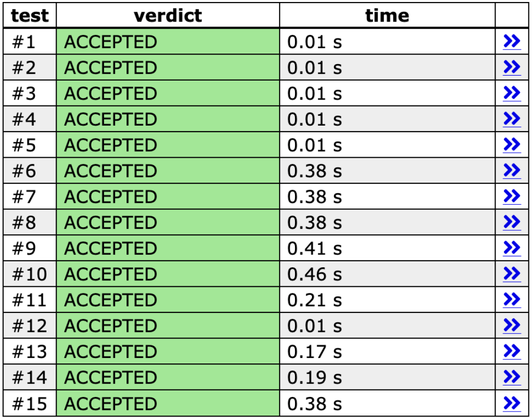
By swapping the sets, if we print out the set’s size as the answer, the result would be incorrect because the subtrees’ sets have been swapped. Therefore, we must maintain the actual size of each set.

Merging Data Structure

By swapping the sets, if we print out the set's size as the answer, the result would be incorrect because the subtrees' sets have been swapped. Therefore, we must maintain the actual size of each set.

```
set<int> color[N];
int sz[N];
void dfs(int u, int p) {
    for(auto v: adj[u]) {
        if(v == p) {
            continue;
        }
        dfs(v, u);
        merge(color[u], color[v]);
    }
    sz[u] = color[u].size();
}
```

Merging Data Structure



test	verdict	time	
#1	ACCEPTED	0.01 s	»
#2	ACCEPTED	0.01 s	»
#3	ACCEPTED	0.01 s	»
#4	ACCEPTED	0.01 s	»
#5	ACCEPTED	0.01 s	»
#6	ACCEPTED	0.38 s	»
#7	ACCEPTED	0.38 s	»
#8	ACCEPTED	0.38 s	»
#9	ACCEPTED	0.41 s	»
#10	ACCEPTED	0.46 s	»
#11	ACCEPTED	0.21 s	»
#12	ACCEPTED	0.01 s	»
#13	ACCEPTED	0.17 s	»
#14	ACCEPTED	0.19 s	»
#15	ACCEPTED	0.38 s	»

YAY, we got AC. Why?
(Source Code)

Merging Data Structure

Why "merge small to large" is work in this situation?

Claim

Each element has to move at most $\mathcal{O}(log(n))$ times.

Proof: When we are trying to merge set A to set B ($|A| \leq |B|$). Then, the new set C's size is at least $|A| + |A| = 2|A|$. Thus, we move the element e for k times. e will be moved to a set that have size at least 2^k. Since the maximum size of the set is n (at root). So each element will be moved at most $\mathcal{O}(\log(n))$ times.

$$\begin{aligned} n &\geq 2^k \\ \log(n) &\geq k \end{aligned}$$

Running time of the algorithm is $\mathcal{O}(n \log^2(n))$

Sack Technique

Problem

Distinct Colors You are given a rooted tree consisting of n nodes. The nodes are numbered $1, 2, \ldots, n$ and node 1 is the root. Each node has a color.
For each node, determine the number of distinct colors present in its subtree.

Sack Technique

Sack technique can "reuse the data structure" while "merge small into large" technique requires lots of data structure.

Sack Technique

Terminology

- "Heavy Child" is a child vertex with the largest subtree size. If there are multiple, pick any.
- "Light Child" is a child vertex which is not a heavy child.
- "Light Edge" is a edge between light child v and its parent u.

Sack Technique

Heavy child can be easily find by DFS.

```
int sz[N], heavy[N];
void dfs(int u, int p) {
    sz[u] = 1;
    for(auto v: adj[u]) {
        if(v == p) {
            continue;
        }
        dfs(u, v);
        sz[u] += sz[v];
        if(sz[v] > sz[heavy[u]]) {
            heavy[u] = v;
        }
    }
}
```

Sack Technique

After we calculate each subtree's size, we try to solve the problem. By using this algorithm:

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

Note: To iterate through every vertices in each subtree, there is a famous approach called "Euler Tour Technique".

Sack Technique

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

First, we define the function above to solve this problem. The meaning of this function is: "We are considering the subtree rooted at u, which has parent p, and `deleting` is a variable that determines whether the updated values from this subtree need to be deleted or not."

Sack Technique

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

We have to delete the updated data of vertex u when u is light child of p. In the other hand, we do not delete the updated data from vertex u when u is the heavy child of p.

Sack Technique

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

Since we did not delete the heavy child's data, the remaining data to update includes the root of the subtree (u) and the data of all children that are not part of the heavy child's subtree.

Sack Technique

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

Now that all the data is inserted, we can store the result in an array. Then, if the subtree's data needs to be deleted, we can simply remove it from the data structure.

Sack Technique

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

In this approach, instead of using set or map as a data structure to count distinct colors, we can just use array to store the data so running time of inserting, erasing, and querying is $\mathcal{O}(1)$.

Sack Technique

```
SACK(u, p, deleting):
    FOR light child v of u:
        SACK(v, u, true)
    IF u is not a leaf:
        SACK(heavy, u, false)
    INSERT(VAL(u))
    FOR light child v of u:
        FOR x in subtree v:
            INSERT(VAL(x))
    // MEMORIZE ANSWER HERE
    IF deleting: // u is light child
        FOR v in subtree u:
            DELETE(VAL(v))
```

(Source Code)

Sack Technique

Number of Insertion

- Every vertex u have to be inserted 1 time when we call SACK(u).
- Every time we try to insert x in subtree v which is light child of u. When SACK(u) is called, the subtree size of u have to be at least 2 times of the size of subtree v.

Thus, each vertex will be inserted at most $1 + \lfloor \log(n) \rfloor$
Number of Deletion: Number of deletion of each vertex is always not exceed the number of insertion which is $1 + \lfloor \log(n) \rfloor$
Number of Query: Query function of the data structure is called exactly n times.
Since the query of this data structure is $\mathcal{O}(1)$. So, the running time of the algorithm is $\mathcal{O}(n \log n)$

Lomsat gelral

Problem (Lomsat gelral)

Too long... Please read in Codeforces.

Can we use "merge small to large" technique or "sack" technique to solve this problem?