Tree - 2

Euler's tour

Euler tour is defined as a way of traversing a tree such that each vertex is added to the tour when we visit it (either moving down from parent vertex or returning from child vertex). We start from the root and reach back to root after visiting all vertices.

It requires exactly **N** vertices to store the Euler tour. Let's understand its approach and algorithm with the help of the following question.

Subtree Query

Problem statement: The tree has 'N' nodes numbered as **1, 2, 3, ..., N**. The topmost node is node-1, i.e., assume that the tree is rooted at node-1. You are also given an integer array **V** of size **N**, where i-th element denotes the weight assigned to the i-th node in the tree.

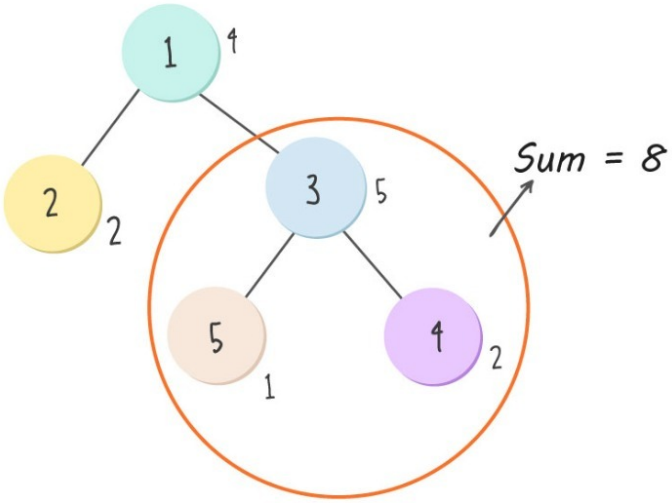
Your friend asked you to perform the following type of queries on the tree and tell him the results.

"1 s w": change the weight of node 's' to 'w'.
"2 s": calculate the sum of weights in the subtree of node 's' (node s inclusive) and return it.

Example:

edges= [[1,2],[1,3],[3,4],[3,5]]

arr = [4,2,5,2,1] , que= 2 3;



Approach: There is a standard problem in which an array is given, and you have to perform two types of queries:

Point update, i.e., updating an element in the array.

Range query, i.e., finding the sum of a given subarray in the array.

The above problem can be solved using a special data structure called a Segment tree.

Run a **dfs** traversal from the tree's root and start a timer with **0**. In each recursion call, increase the timer by **1**. For each node, keep track of the timer when the recursion starts and ends. Let the timer be at **t1** when a recursion call on node **x** is made; it will recursively process all its children and their children too. After processing the subtree of node **x**, let the times be at **t2**.

Maintain an array where the i-th element denotes a node's weight for which starting timer is **i**.

In such a way, the sum of subtree under node **s** will be the sum of a subarray in the range **[start_time_of_s , end_timer_of_s]**.

Algorithm:

- Implement a segment tree and all its necessary functions **(build, update, query)** as standard ones.
- Start a time at **t = 0**.

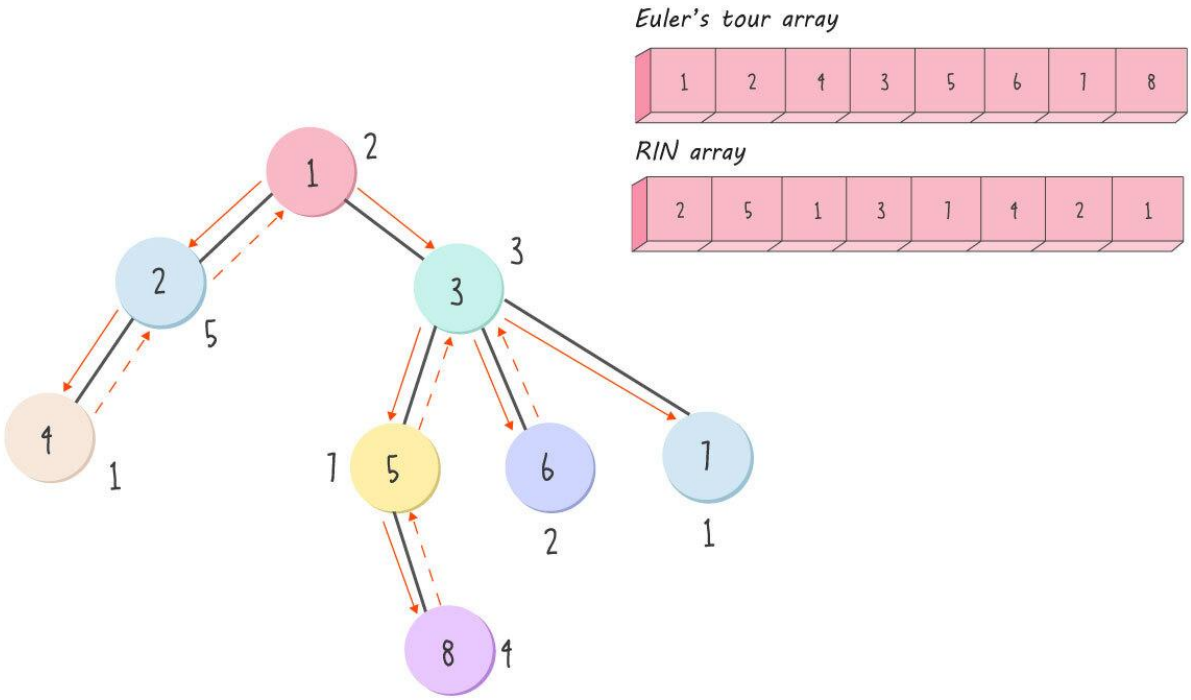
Implement a dfs to calculate both timers for each node.

- Function accepts two parameters: current node **x** and its parent **p**.
- Store starting timer of node **x** as **t**.
- Increment timer **t** with **1**.
- For all the neighbours of the current node.
- If a neighbour is not the parent **p**.
- Call dfs on the neighbour node.

- Store ending timer of node **x** as **t**.

Implement the main function.

- Run the **dfs** from the root node and create a segment tree on **rin** array.
- Process the queries one by one.
- If the query is of **type-1**, call the update function of the segment tree.
- If the query is of **type-2**, call the query function of the segment tree and return the result.



Code:

```
#include<bits/stdc++.h>
#define int long long

using namespace std;
```

```
const int N = (int)(2e5+5);
vector<int> tree[N];
int val[N];
int in[N], out[N], rin[N];
int T = 0;

void dfs(int s, int p){
      in[s] = T++;
      rin[in[s]] = val[s];
      for(int v: tree[s]){
            if(v!=p) dfs(v, s);
      }
      out[s] = T;
}

int st[4*N];

void build(int beg, int end, int pos){
      if(beg == end){
            st[pos] = rin[beg];
            return;
      }

      int mid = (beg + end) / 2;
      build(beg, mid, 2*pos+1);
      build(mid+1, end, 2*pos+2);
      st[pos] = st[2*pos+1] + st[2*pos+2];
}

int query(int beg, int end, int ql, int qr, int pos){
      if(ql<=beg and qr>=end){
            return st[pos];
      } else if(qr<beg or ql>end) return 0;
      int mid = (beg + end)/2;

      return query(beg, mid, ql, qr, 2*pos+1) + query(mid+1, end, ql, qr,
2*pos+2);
}
```

```
void update(int beg, int end, int idx, int val, int pos){
	if(beg == end){
		st[pos] = val;
		return;
	}
	int mid = (beg + end)/2;
	if(idx<=mid){
		update(beg, mid, idx, val, 2*pos+1);
	} else{
		update(mid+1, end, idx, val, 2*pos+2);
	}

	st[pos] = st[2*pos+1] + st[2*pos+2];
}

signed main(){
	int n, q;
	cin>>n>>q;
	for(int i=1; i<=n; i++) cin>>val[i];

	for(int i=0; i<n-1; i++){
		int u, v;
		cin>>u>>v;
		tree[u].push_back(v);
		tree[v].push_back(u);
	}

	dfs(1, 0);

	build(0, n-1, 0);
	while(q--){
		int type;
		cin>>type;

		if(type == 1){
			int s, v;
			cin>>s>>v;
			update(0, n-1, in[s], v, 0);
		} else{
			int s;
```

```
                cin>>s;
                cout<<query(0, n-1, in[s], out[s]-1, 0)<<"\n";

            }
        }

        return 0;
}
```

Time Complexity:

O(N + Q*log(N)), where **N** and **Q** are the number of nodes and queries, respectively.

Both update and range sum queries take **O(logN)** time in a segment tree. Hence, overall time complexity becomes **O(N + Q * log N)**.

Space Complexity:

O(N), where **N** is the number of nodes in the tree.

Storing the edges in the form of a tree takes space **O(N)**. An array parent of size **N** is also maintained. Hence, overall space complexity becomes **O(N)**.

Introduction to MO's algorithm

The idea of MO's algorithm is to pre-process all queries so that the result of one query can be used in the next query. It is an **offline algorithm** (when queries don't

require any update). E.g. when each query asks to find the mode of its range (the number that appears the most often). For this, each block would have to store the count of each number in it in some sort of data structure, and we can no longer perform the merge step fast enough anymore. Mo's algorithm uses a completely different approach, that can answer these kinds of queries fast, because it only keeps track of one data structure, and the only operations with it are easy and fast. To understand this algorithm let's take an example of a problem.

Problem statement:

We are given a tree and a set of query ranges, we are required to find the sum of every query range.

Approach:

First of all we convert the tree into array using Euler's tour and after that we will apply MO's Algorithm.

Algorithm:

Let **a[0...n-1]** be the array which is formed after applying Euler's tour and **q[0..m-1]** be the query array.

1. All queries should be sorted in such a way that queries with **L** values ranging from **0** to **sqrt(n) – 1** are grouped together, followed by queries with **L** values ranging from **sqrt(n)** to **2*sqrt(n) – 1**, and so on. Within a block, all queries are ordered in increasing order of **R** values.

2. Process each query one at a time, making sure that each one uses the sum computed in the preceding query.

- Let **sum** be the total of the previous queries.
- Remove any remaining entries from the preceding query. If the prior query was **[0, 8]**, and the current question is **[3, 9]**, we deduct **a[0]**, **a[1]**, and **a[2]** from the sum.
- To the current query, add new elements. We add **a[9]** to total in the same example as before.

The amazing thing about this technique is that the index variable for **R** changes its value at most **O(n * sqrt(n))** times throughout the run, It's because for a particular block we move **R** only in one direction. So it can travel a total distance of **O(N)**. Total blocks are sqrt**(N)**. So **N*sqrt(N).**

For **L**, in each query, it can go at max by **sqrt(N)** (within the same block). So **Q** queries mean a total of **Qsqrt(N)**

while the same is true for **L**. All these bounds are possible only because the queries are sorted first in blocks of **sqrt(n)** size.

Preprocessing takes an **O(m*Logm)** amount of time.

All queries are processed in **O(n * sqrt(n)) + O(m * sqrt(n)) = O((m+n) * sqrt(n))** time.

Code for MO's algorithm:

```
int block;
struct Query
{
    int L, R;
};

bool compare(Query x, Query y)
{

    if (x.L/block != y.L/block)
        return x.L/block < y.L/block;
    return x.R < y.R;
}

void queryResults(int a[], int n, Query q[], int m)
{
    block = (int)sqrt(n);
    sort(q, q + m, compare);
    int currL = 0, currR = 0;
    int currSum = 0;

    for (int i=0; i<m; i++)
    {
        int L = q[i].L, R = q[i].R;
        while (currL < L)
        {
            currSum -= a[currL];
            currL++;
        }
        while (currL > L)
        {
            currSum += a[currL-1];
            currL--;
        }
        while (currR <= R)
        {
```

```
            currSum += a[currR];
            currR++;
        }
         while (currR > R+1)
         {
            currSum -= a[currR-1];
            currR--;
         }
        cout << "Sum of [" << L << ", " << R << "] is "  << currSum << endl;
    }
}
```

Important Points to Consider:

1. All requests are known ahead of time, allowing them to be preprocessed. It won't function in situations when update operations are mixed along with **sum** queries.
2. If in a problem the result of the previous query can be used to solve the next query then this type of problem can be solved using **MO's approach**. Another example is the term **maximum** or **minimum**.

Time Complexity: O((m+n) * √n) where **m** is a number of queries and **n** is the size of the array.

Space complexity: O(n+m), where **m** is the number of queries and **n** is the size of the array.