NOI.PH Training: Graphs 3

Trees

Robin Yu, Kevin Atienza

Contents

1 Introduction

We'll discuss more advanced data structures and algorithms related to trees. There are two flavors in which "trees" appear in this module:

- Problems whose setting is a tree.
- Data structures that are based on (rooted) trees.

1.1 How to read this module

Many things discussed here are problem-solving techniques which can be used as black boxes to solve other problems. That's fine, although a deeper understanding of the techniques is recommended so that you can debug your code better, and you can reuse the underlying ideas in other problems.

It's very tempting for you to just read everything continuously in order, and just learn the algorithms directly and how to apply them, without having to think for yourself. However, what will happen is that you'll just come to view them as those advanced techniques handed down from on high, not knowing how in the world anyone can come up with it. That's not the way to get the most out of this module! Learning the specific techniques is easy enough, but that's only half of it.

If you want to get the most value out of this text, I recommend stopping at every point and trying to think of what comes next, especially if prompted by the text.

Also, before actually reading on, it might be a good idea for you to read the problems that will be used as motivation for the techniques, and then try solving them first. Look for them at the beginning of most sections; they're marked as **Problem X.X**. Avoid reading the surrounding text as you do so.

If you get stuck, you can look at the section title to see the name of the technique. Even if you don't know the details yet, maybe you can gain some insights just from the names alone, which I think are very suggestive.

Finally, do the exercises before proceeding since they're part of the text's flow; some later parts will only make sense (or make sense better) if you've done them.

If you get stuck, ask in Discord for help! Some of these topics are not easy, so it's perfectly acceptable to ask.

The following lessons will be more valuable if you've at least attempted the problems and gained some partial insights.

2 (Update, Query) ↔ (Query, Update)

Consider the following problem:

> **Problem 2.1.** You have an array p of length n, all elements initially zero. You need to support q operations, each of one of the following types:
>
> 1. **Range update**. Given l, r and v, add v to each of the $p_l, p_{l+1}, \ldots, p_r$.
> 2. **Point query**. Given x, report p_x.

This is a relatively straightforward problem. Think about it for a bit before proceeding.

If you started thinking of using some segment trees with lazy propagation or juggling two Fenwick trees, you're probably correct. However, they are overkill for this task.

In fact, we can solve this task with just a single Fenwick tree solving the classic range sum query problem!

How? Well, we can convert those two kinds of operations to the following:

1. Given l, r and v, add v to p_l and $-v$ to p_{r+1}.
2. Given x, report $p_1 + p_2 + \cdots + p_x$.

You should be able to see immediately why this works; this is simply a variation of the prefix sum technique.

The runtime is still $\mathcal{O}(n + q \log n)$, the same as if you had built the segment tree with lazy propagation. But it's easier to code, and the constant factor is probably significantly lower, so this approach is really better.

This is important because in a contest setting, you usually don't want to spend time coding a complicated solution when a simpler one suffices. Here, the simple approach works, and hence we should use that one instead.[1] Of course, if you've already coded the segment tree, then it might be better to just use that.

Note that this is not always applicable. For example, it will not work if the query instead asked to *set* every value in a range to v, instead of simply adding. Generally, for this to work, the operation involved needs to have an inverse operation; for example, add and subtract, or bitwise XOR with itself.

2.1 As a more general trick

This idea applies to more than just this particular problem. It's more of a technique, or a way of thinking.

What is this "technique", exactly? Well, it's somewhat tricky to explain, so I'll try to explain it by way of examples. Also, the technique doesn't have a name, so I'll invent one for it: the **otherside technique**. (Name suggestions welcome!) So we can now say that we used the otherside technique earlier to turn range updates and point queries to point updates and range queries, respectively.

[1] Doubly so if segment trees are still not second nature to you!

2.1.1 Queries as receptacles

First, let's look at the following problem:

Problem 2.2. Let b be a binary array of length n, all initially zero. You need to process q operations of two types:

- Given i, flip b_i.
- Given i and j, find the number of "connected components" of 1s in the subarray $[b_i, \ldots, b_j]$.

Again, I encourage you to think about it before moving on.

There's a nice solution involving counting the number of subarrays of the form $[0, 1]$ and $[1, 0]$. That's perfectly acceptable, though it doesn't use the otherside technique so we won't discuss it.

Exercise 2.1. Find this solution involving subarrays of the form $[0, 1]$ and $[1, 0]$.

Instead, let's think differently. The number of components in $b_{[i,j]}$ is more-or-less the number of components in $b_{[0,j]}$ minus the number of components in $b_{[0,i)}$,[2] so without loss of generality, let's assume that all queries are prefix queries. That kinda simplifies things, but not by much—it's still mostly the same problem.

Next comes the key idea of the otherside technique: *think of queries as receptacles*. Instead of thinking of a query as something that tells us to do some computation, let's turn them into actual entities in their own right. In other words, we *concretize* them.

Since we're only querying on prefixes, the unique possible queries are $b_{[0,0]}, b_{[0,1]}, \ldots, b_{[0,n]}$. Now, we think of them as receptacles that hold the answers to the queries. Let a_i be the answer to the query on $b_{[0,i]}$. Since the array is initialized at 0, we have $a_i = 0$ initially.

Since we're directly storing the answers to queries, answering a (prefix) query becomes a straightforward lookup to the appropriate a_i. All that remains is to update them when we flip a bit, say bit i:

- If bit i is being flipped from 1 to 0, then we're basically cutting off a connected component into two separate pieces. The number of components for each prefix below i won't be affected, but the number of components for each prefix above i increases by 1.

  The exception is if one or both neighbors of bit i is 0, which you need to handle a bit differently. We also need to think about what happens to a_i itself. I'll leave it to you to figure these all out.

- If bit i is being flipped from 0 to 1, then we basically do the opposite of the previous case. In particular, among other things, we decrease each a_j for $j > i$ by 1. (Though again, something different happens if bit i has a neighbor of 1.)

But now, notice that we've reduced the operations to just point queries and range updates on the array $[a_0, a_1, \ldots, a_n]$—these are just the segment tree operations! So we've just solved the problem with $\mathcal{O}(\log n)$ time per operation (and $\mathcal{O}(n)$ preprocessing).

Furthermore, as we've just seen above, we can switch from range updates and point queries to point updates and range queries (using the otherside technique again).

[2] We use the notation $b_{[i,j]} := [b_i, \ldots, b_j]$, $b_{[i,j)} := [b_i, \ldots, b_{j-1}]$, etc.

Exercise 2.2. The number of components in $b_{[i,j]}$ is not quite equal to the number of components in $b_{[0,j]}$ minus the number of components in $b_{[0,i)}$. What's wrong? Also, can you fix the solution to account for this?

Exercise 2.3. Fill in the missing details of this solution.

Exercise 2.4. Solve the following problem in $\mathcal{O}(n+q\log n)$ time using the otherside technique:

> *Let b be a binary array of length n. You need to process q operations of two types:*
>
> - *Given i, flip b_i.*
> - *Given i and j, find the sum of the squares of the sizes of the connected components of 1s in the subarray $[b_i, \dots b_j]$.*

In fact, even the idea of "prefix sums" is a basic example of the otherside technique. It's just too basic that we don't think of it that way.

2.1.2 Updates as receptacles

For another example, let's consider the following problem:

Problem 2.3. Congressman Armando Rugas is a well-respected politician who has won every election he's participated in. As a thank you to the voters, he goes around Quezong city to give gifts to the residents.

Quezong city has n houses, each containing a number of residents. Each resident has an age, which is given as a number between 1 and 10^{10}. (The age is given in seconds.)

You need to process q operations:

- Given a and g, Armando gives a gift of g pesos to every house that satisfies two properties: (1) it has a resident whose age is $\leq a$, and (2) it has a resident whose age is $\geq a$.
- Given i, find the total amount that has been gifted to the ith house so far.

There are several insights to be found in this problem. You don't have to find them all to solve it. Different sets of insights yield distinct solutions. Here, I'll describe a set of insights that can be used in conjunction with the otherside technique to solve it.

As usual, please try solving it first before proceeding.

First, because of the nature of the gift-giving, the only things that matter per house are the ages of the youngest and the oldest residents in it. Let's call these x_i and y_i for the ith house, respectively, with $x_i \leq y_i$. We can now say that Armando gives g pesos to the ith house iff $x_i \leq a \leq y_i$.

A house is fully characterized by two numbers (x_i, y_i), so we can plot them as *points on the plane*. (This is another sort of "technique": Plot your data in Cartesian space to take advantage of humans' innate spatial intuition.) Each of these points holds a value, the total amount gifted to the house. An update corresponds to increasing this value by g for every point (x_i, y_i) satisfying $x_i \leq a$ and $y_i \geq a$. The region of all points (x, y) satisfying $x \leq a$ and $y \geq a$ is a sort of "quarter-plane" with corner (a, a)—specifically, they're all the points located top-left

of point (a, a).

At this point, the most natural way to complete the solution is to just simulate the operations directly, say via a 2D segment tree, which supports operations on subrectangles. However, it takes some of effort to code, and if the implementation is too simple, it may even be too slow ($\mathcal{O}(\log^2 n)$ per operation).

Instead, we use the otherside technique: *think of updates as receptacles*. Instead of storing the total amounts at the houses, we store them at the *updates* instead. Specifically, for each a between 1 and 10^{10}, we store the total amount Armando has gifted with this a. Let's denote this by t_a. Then the operations turn into the following:

- an update corresponds to just increasing t_a by g, and
- a query for house i consists of adding up all the t_as for the as in the range $[x_i, y_i]$.

But we now recognize this as the segment tree operations again! And you don't even need lazy propagation!

Thus, by using the otherside technique, we've found a solution that runs in $\mathcal{O}(q \log(\text{max age}))$ (where $(\text{max age}) = 10^{10}$).

Exercise 2.5. Update this solution to run in $\mathcal{O}(n + q \log q)$ instead of $\mathcal{O}(n + q \log (\text{max age}))$.

Exercise 2.6. Solve the following problem in $\mathcal{O}((n+q) \log n)$ time using the otherside technique:

The premise is the same as Problem 2.3, but the operations are now different.

- *Given a, b and g, Armando gives a gift of g pesos to every house that satisfies two properties: (1) it has a resident whose age is $\leq a$, and (2) it has a resident whose age is $\geq b$.*
- *Find the house that has been given the maximum total amount so far. If there are multiple, output the lowest-indexed one.*

2.1.3 Summary

We can roughly think of the otherside technique as basically an instance of *looking at it from another perspective*—after all, we're switching the roles of updates and queries, or we're focusing on the queries (or updates) as concrete objects rather than the actual concrete objects mentioned in the problem, and so on.

All in all, it's really hard to make precise what we mean by the "otherside technique", but I hope the examples above illustrate it well!

3 Flattening a Tree

Many problems involving queries on subtrees of a tree can be turned fairly straightforwardly into problems involving queries on subarrays of an array.

Here is a simple example. Consider the following problem:

Problem 3.1. You have a rooted tree with n vertices. Each has a non-negative integer value p_i; you need to support q queries, each of one of the following types:

1. **Subtree query**. Given a vertex v, report the sum of the values of all descendants of v (including v itself).
2. **Point update**. Given a vertex v, add x to p_v.

As usual, I encourage you to try solving this first before proceeding.

There are some complicated ways to solve this, possibly trying to work directly on the tree, but this problem becomes much simpler if we "flatten" the tree first.

At this point, I think it's worth trying to solve it again before proceeding, especially if you still haven't solved it yet, but also if your solution doesn't involve flattening the tree (whatever that means). You can take the suggestive name "flattening the tree" as a hint. What does it mean to flatten a tree? Do you know how of ways to flatten a tree? You probably already do, you just haven't thought of them that way! If you still haven't got it, I'll give a final hint in the footnote which you can look at before you proceed.[3]

Anyway, to see what "flattening the tree" means, consider the tree shown in Figure 1:

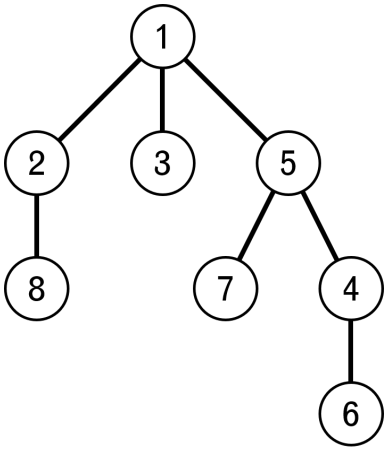

Figure 1: Rooted tree we will use as an example

Now, let's write out the pre-order traversal of this tree[4] (Figure 2):

1	2	8	3	5	7	4	6

Figure 2: Pre-order traversal of the rooted tree we used as an example

[3] graph traversals

[4] Note that the exact pre-order traversal can differ if the subtrees are traversed in a different order, but that does not change anything substantial in this algorithm. The same is true for the Euler tour traversal discussed later. Here, we just go from left-to-right in the illustration, even if, for example, 4 is smaller than 7.

Notice something wonderful? We need to do subtree queries—where did the subtrees end up in the pre-order?

Notice that every subtree of the tree is a consecutive range on this pre-order traversal array! For example, the subtree rooted at vertex 5 is represented by the subarray $[5, 7, 4, 6]$ in the pre-order traversal.

Hence, this problem is transformed into the classic range sum query problem; the sum of a subtree is simply a sum of a range, and changing the value of a vertex is the same as changing the value of an element in an array. So, we can solve this problem with just a single Fenwick tree in $\mathcal{O}(n + q \log n)$!

Combining this trick with the otherside technique (illustrated in section 2), it is clear how we can solve this problem if the operations were instead changed to:

1. **Point query**. Given a vertex v, report the value p_v.
2. **Subtree update**. Given a vertex v, add x to the values of all descendants of v (including v itself).

Note that the solution also requires only one Fenwick tree; think about how we might solve it!

Note that, if we are willing to use segment trees with lazy propagation, or to juggle two Fenwick trees, then we can even solve the problem where both types of operations are on subtrees:

1. **Subtree query**. Given a vertex v, add x to the values of all descendants of v (including v itself).
2. **Subtree update**. Given a vertex v, report the sum of the values of all descendants of v (including v itself).

These queries look complicated at first glance, but if we forget about subtrees and just think about subarrays, then the solution is elucidated.

This is a general principle that can be applied to many problems; if you have already simplified the problem, think in terms of the simplified version instead of the harder original problem!

This "flattening the tree" approach is something useful to try on any problem that asks about queries on subtrees: subtree sum, subtree minimum/maximum, and so on.

There are times, however, when solving the problem is actually *easier* on subtrees than on subarrays. For example, it's a bit more straightforward to solve the static "count how many distinct values are in each subtree" problem without flattening the tree, instead using the $\mathcal{O}(n \log^2 n)$ smaller-to-larger merging approach. (Another such problem using that method, if you are not too familiar with it, is this one from Codeforces: 600E - Lomsat gelral). So, the method of flattening the tree is not universally the best, but it is always good to try it nonetheless.

Exercise 3.1. You have a rooted tree with n vertices. Each has a non-negative integer value p_i; you need to support q queries, each of one of the following types:

1. **Point update**. Given a vertex v, add x to p_v.
2. **Root path query**. Given a vertex v, report the sum of the values of all *ancestors* of v (including v itself).

Hint: otherside technique

3.1 Euler Tour Technique

Another way of flattening the tree involves something called an **Euler tour** of the tree, which is not the same an an Eulerian circuit or path, although both are named after the same guy.

There are a number of conflicting definitions of the Euler tour technique.[5] The varying definitions are all useful to learn, as they can be used to solve different problems. However, these definitions are based on the same overarching idea.

We will discuss three definitions here. We will use the following simple tree to illustrate the subtle differences between them:

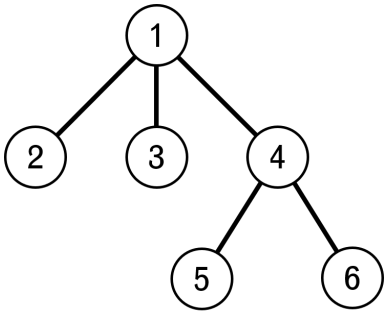

Figure 3: Rooted tree which will illustrate the differences between the three types of Euler tour techniques

All definitions are based on the "Euler tour", which is basically the following traversal:

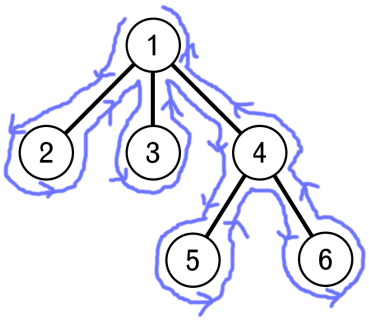

Figure 4: The Euler tour of the tree in Figure 3.

You'll notice that it's very similar to the DFS or pre-order traversal. The only thing that differs is that you're not only recording the first occurrence of everything you encounter, but basically all occurrences. So some of the nodes or edges may appear more than once, for example.

3.1.1 Edge ETT

The first definition, which will be referred to as **Edge ETT**[6], is the ordered list of edges that we will encounter if we do a depth-first search from the root, in both directions (going

[5] These definitions are also outlined here.
[6] not official terminology

down and retreating upwards). For the above tree, one such Edge ETT may be the following:

$$[(1,2),(2,1),(1,3),(3,1),(1,4),(4,5),(5,4),(4,6),(6,4),(4,1)]$$

Now, there are some things we can do this this list. If it is the edges that have weights, instead of the vertices, and we need to do subtree queries, this is a very natural way to solve that problem; notice that subtrees still turn into subarrays in this array.

There is also a trick for *path sums* on the Edge ETT. (This trick does not work for minimum/maximum or other non-invertible operations; you should try Heavy–Light decomposition instead, which is discussed in section 5.) Let's state the operations:

Problem 3.2. Given a rooted weighted tree with n nodes, process q operations:

- **Edge update**. Given an edge and a value v, increase the weight of the edge by v.
- **(Root) path update**. Given a node i, find the weight of the path from the root to i.

I encourage you to think about how to apply the Edge ETT to solve this. A hint is that it's very similar to Exercise 3.1.

First, let's look at the first occurrence of every node in the Edge ETT:

$$[(\underline{1},\underline{2}),(2,1),(1,\underline{3}),(3,1),(1,\underline{4}),(4,\underline{5}),(5,4),(4,\underline{6}),(6,4),(4,1)]$$

Next, when we update an edge, think about which paths to the root will be updated. Where are these nodes located in the edge ETT? Once you understand this, I think the full solution becomes somewhat apparent (in conjunction with earlier techniques, of course).

Exercise 3.2. Solve Problem 3.2 with the Edge ETT.

3.1.2 Vertex ETT

The second definition, which will be referred to as **Vertex ETT**[7], is very similar to the Edge ETT, in that it is the list of vertices we will encounter if we do a depth-first search from the root. We also do it "in both directions"; this time when we enter it for the first time, and when we leave it for the last time. (This is naturally handled by a recursive implementation of depth-first search.)

For the example tree, we have the following Vertex ETT:

$$[1,2,2,3,3,4,5,5,6,6,4,1]$$

The usage of the Vertex ETT is very similar to that of the Edge ETT, but it is better suited if the vertices are the ones with weights. Notice that the descendants of a particular vertex are particularly those enclosed by the two occurrences of that vertex on the list, so that subtrees are also easy to query here. Of course, we should be careful not to double count on queries and updates. For example, we could just update the first occurrence of the vertex on the list.

Path sums are also possible to do here, which is again suited if the vertices are the ones with values. The details are very similar to the Edge ETT; you are encouraged to try to work them out on your own. (Actually, it's basically the same as the solution to Exercise 3.1, just with extra stuff in the array.)

[7] also not official terminology

3.1.3 LCA ETT

The final definition, which I will refer to as **LCA ETT**, of course not official terminology, is also a list of vertices, but it is a bit different from the Vertex ETT. It is the list of vertices we will encounter if we do a depth-first search from the root, but we list a vertex down every time we enter it, whether from below or from above.

For the example tree, we have the following LCA ETT:

$$[1, 2, 1, 3, 1, 4, 5, 4, 6, 4, 1]$$

3.1.4 Things we can do with ETT

What can we do with ETTs? Well, for a start, it's a tree-flattening technique, so for example you can use it to do most things you can do with the pre-order traversal. However, there are a few things that are easier to think about with ETTs than with pre-order. Here are a couple:

Exercise 3.3. Describe how *rerooting* the tree (that is, choosing some vertex v to be the new root) affects each of the ETTs. Which ones are easy to update?

Exercise 3.4. Describe what happens if we *cut* a subtree, that is, choose a node and remove it and all its descendants from the tree. Which ETTs are easy to update?

Exercise 3.5. Similarly, describe what happens if we *insert* a new subtree as a child of some existing vertex.

Finally, ETT can also be used to solve LCA queries in a different way, which we will see in the next part.

3.2 Lowest Common Ancestor → Range Minimum Query

The reason we call the last ETT the "LCA ETT" is because this provides another way of computing the lowest common ancestor of any two nodes, besides the standard binary lifting method that you might already know.

To begin with, consider the LCA ETT and the two nodes u and v you wish to find the LCA of. Find u and v in the LCA ETT. (There are many occurrences; choose any one.) WLOG, assume that u appears before v. What can you say about the LCA?

Exercise 3.6. Show that the LCA appears between u and v in the LCA ETT.

So we know that the LCA appears between u and v...this already smells like range queries. But how do we identify the LCA among these nodes? Can we even identify any common ancestor between them?

As it turns out, it's pretty easy:

Exercise 3.7. Show that the node with the least depth between u and v is the LCA.

Hence, we have reduced LCAs to range minimum queries!

This approach to the LCA problem is quite nice. If we consider the traditional binary lifting approach, it needs $\mathcal{O}(n \log n)$ time and memory to preprocess and takes $\mathcal{O}(\log n)$ time per query. On the other hand, this approach:

- Needs only $\mathcal{O}(n)$ time and memory to preprocess if we solve the resulting range minimum query problem with a segment tree.
- Takes only $\mathcal{O}(1)$ time per query if we solve the resulting range minimum query problem with a sparse table.

While keeping all other complexities asymptotically the same! So regardless of which method we choose to solve the underlying range minimum query problem, it results in an algorithm that is asymptotically better in some way.

Of course, that doesn't necessarily mean you should *always* use this reduction; the binary lifting method is often easier to implement, and the difference in runtime usually doesn't end up making a big difference anyway. (And maybe you need binary lifting for some other part of your solution.) But the differences are there, and it is good to keep note of them in case it turns out to be useful.

It is possible to exploit this reduction further to get an $\mathcal{O}(n)$ preprocess, $\mathcal{O}(1)$ query algorithm for the lowest common ancestor problem, and in fact also for the general static range minimum query problem. Its application to LCA/RMQ is not really essential in competitive programming, but the general technique behind it may be of use to other problems. It's also of theoretical interest. You are encouraged to read on it if you like. It's called the *method of four Russians*.[8] However, although we won't describe the four Russians trick here, we'll prove an important aspect of the reduction: the fact that *the LCA and RMQ problems are equivalent*, computationally. We've already shown part of it with the reduction above. We'll show the other part in subsection 4.2.

[8] It's a technique developed by four people working in Moscow at the time—Arlazarov, Dinic, Kronrod, and Faradžev—although I don't actually know if they're all Russians.

4 Cartesian Tree

We'll motivate the idea behind the so-called Cartesian tree with this problem:

Problem 4.1. There are n mountains in a mountain range whose heights are $h_1, h_2, \ldots, h_n$. The heights are distinct.

Suppose you're on a mountain. You can only glide towards another mountain if it has a strictly smaller height, and there's no intermediate mountain that's higher than the mountain you started on. Let $g(i)$ be the maximum number of glides you can do, starting from mountain i.

Compute $g(1), g(2), \ldots, g(n)$.

As usual, think about it first before moving on.

Let's look at some mountain, i with $1 \le i \le n$. Which mountains can you glide to next? You can't glide to a mountain taller than i. However, you also can't glide to a mountain shorter than i if there's a taller one in the way. In particular, we can't glide to any mountain to the left of the first taller mountain to the left of i, and the same is true for the right side. However, there are no other restrictions. (Why?) So if these first taller mountains are L_i and R_i, respectively, then we can glide to precisely the mountains between L_i and R_i, *exclusive* (and excluding i too). If the said mountain doesn't exist, we just define $L_i = 0$ or $R_i = n + 1$.[9]

Among these mountains, which one should we glide to? Suppose we go left. Then we can't glide to any mountain to the right of i anymore, so we're left with the mountains in the interval (L_i, i). But among these mountains, gliding to the tallest mountain gives us the most options later since you can glide to any of them next. Therefore, if we go left, then we want to glide to the tallest mountain among them. A similar thing is true if we go right. So we're left with two options, one for each side, and we can simply go to the better one among them.

Let these tallest mountains be ℓ_i and r_i, respectively. (One or both may not exist.) Then we have the recurrence

$$g(i) = \max(g(\ell_i) + 1, g(r_i) + 1, 0).$$

If ℓ_i or r_i doesn't exist, we can define its g to be $-\infty$ so that the recurrence still works.

In fact, we've just described the Cartesian tree! We can think of ℓ_i and r_i as the left and right children of node i, and it isn't hard to see that they form a tree!

Exercise 4.1. Prove that the ℓ_i and r_i pointers form a rooted tree whose root is the index containing the maximum value.

Phrasing things in terms of the Cartesian tree, we now see that $g(i)$ is simply its *height* in the Cartesian tree. Thus, if we can find the Cartesian tree, then we can solve this problem easily!

So in what follows, we'll figure out how to compute it.

By the way, some people take "Cartesian tree" to mean the same thing as a *treap*. Although a treap has the form a Cartesian tree, I think it's better to think of them as separate concepts:

- In a Cartesian tree, we're given a sequence of values, and we build the tree determined by those values. There's no guarantee that the tree will be balanced; its height can be $\Theta(n)$.
- In a treap, we're also given a sequence, but we *choose* the values—"priorities" in treap terms. The sequence itself don't really influence this choice. Therefore, we're free to choose the priorities so that the tree is more-or-less balanced.

[9] This is equivalent to putting sentinel mountains on both sides that are infinitely tall.

Exercise 4.2. What goes wrong when the heights are not distinct? Can you still solve the problem?

4.1 Constructing the Cartesian tree

We would now like to construct the Cartesian tree of a sequence $[a_1, \ldots, a_n]$. Note that constructing the Cartesian tree is equivalent to computing ℓ_i and r_i for each i. [10]

We're now actually partway into the construction! By definition, ℓ_i and r_i refer to the largest values in the intervals (L_i, i) and (i, R_i), so we're basically done if we can compute in L_i and R_i for each i, since we can just do range minimum queries to get ℓ_i and r_i. Even better, we can bypass these RMQs with the following insight:

Exercise 4.3. Show that the parent of i in the Cartesian tree is either L_i and R_i, whichever one refers to the smaller value.

So we really only need to compute the L_is and R_is.

Problem 4.2. Given an array $[a_1, a_2, \ldots, a_n]$, compute L_i and R_i for each i, defined as:

- L_i is the largest index $< i$ such that $a_{L_i} > a_i$, or 0 if it doesn't exist.
- R_i is the smallest index $> i$ such that $a_{R_i} \geq a_i$, or $n + 1$ if it doesn't exist.

As usual, attempt to solve it first. It can be solved with techniques you already know.

First, we can focus on just the L_is since the R_is are just a mirror image; we can do our computation of L_is but mirrored.

Next, we can actually do some sort of binary search! Notice that the following values are increasing:

- a_{i-1}
- $\max(a_{i-2}, a_{i-1})$
- $\max(a_{i-3}, a_{i-2}, a_{i-1})$
- $\ldots$
- $\max(a_2, \ldots, a_{i-2}, a_{i-1})$
- $\max(a_1, a_2, \ldots, a_{i-2}, a_{i-1})$

Exercise 4.4. Show that L_i is the largest index such that $\max(a_{L_i}, \ldots, a_{i-1}) > a_i$.

Since we can do range max queries in $\mathcal{O}(\log n)$ time each, we can do this binary search in $\mathcal{O}(\log^2 n)$ time with a segment tree!

This can be improved to $\mathcal{O}(\log n)$:

[10] Note that although our motivating problem has distinct values, we don't assume this in general. A "Cartesian tree" can still be built. However, if there are duplicates in the sequence, then the Cartesian tree isn't quite well-defined. One way to define a Cartesian tree is as follows: ℓ_i and r_i refer to the *rightmost* largest values in (L_i, i) and (i, R_i), respectively.

Exercise 4.5. Improve this to $\mathcal{O}(\log n)$ by using the segment tree for range max queries in a different way.

Hint: Traverse the tree starting at leaf i, and look for L_i somehow.

All in all, we get an $\mathcal{O}(n \log n)$ solution. That's perfectly acceptable, and has the added benefit of still being applicable in the dynamic case, where the a_is may change. However, in the static case, we can do better: we can do it in $\mathcal{O}(n)$!

And in fact, we only need a stack, and a single pass across the array from left to right! I'll leave it to you to solve as an exercise:

Exercise 4.6. Solve Problem 4.2 in $\mathcal{O}(n)$ time using a stack.

Hint: The stack should maintain the set of nodes that may still be the L_i for some i in the future. In particular, the stack always contains a decreasing subsequence of a.

4.2 Range Minimum Query → Lowest Common Ancestor

The Cartesian tree allows us to prove the converse to subsection 3.2; we can reduce RMQ to LCA! It's basically a straightforward consequence of the following theorem, which should be intuitive if you really understand what the Cartesian tree means:

Theorem 4.7. Let $A = [a_1, \ldots, a_n]$ be an array. Then the minimum of $a_{[i,j]}$ is a_k, where k is the LCA of i and j in the Cartesian tree of A.

The proof is quite straightforward! So I'll leave it to you as an exercise.

Exercise 4.8. Prove Theorem 4.7.

5 Heavy–Light Decomposition

Consider the following problem:

Problem 5.1. You have a tree with n vertices. Each has a non-negative integer value p_i; you need to support q operations, each of one of the following types:

1. **Path query**. Given two vertices u and v, report the sum of values of the vertices on the unique path from u to v.
2. **Path update**. Given two vertices u and v, set to x the value of every vertex on the unique path from u to v.

There is the straightforward solution which runs in $\mathcal{O}(nq)$—for each of the q queries, use a DFS or BFS to determine the path between u and v, and then either sum them up or add to each of them in a total of $\mathcal{O}(n)$ time, giving $\mathcal{O}(nq)$ overall.

Of course, we are not really interested with just the straightforward solution (although it is often useful to code it anyway for partial points as it is easy to write it in less than ten minutes). We want to know if there is something *better*.

Try finding a faster solution yourself first! If you don't end up solving it, it's okay, since you'll have a better understanding on why the task is not so straightforward, especially given the techniques we already know.

A very good approach with many complicated problems is to first think of solving some simpler, special case of the problem. Let's think about a simpler graph. What's the simplest possible type of graph? Well, you might come up with a line, with all the vertices neatly arranged like so:

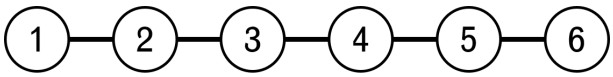

Figure 5: Line graph

Can we solve this problem quickly? Well, in fact, it's not that hard—just forget that this is a graph in the first place, and just treat all the queries as if they were on an array! (Make sure to watch out for $u > v$, since we can start from the *right* here!) This is now the classic range sum query problem which can be solved with a segment tree with lazy propagation. It is doable in $\mathcal{O}(n + q \log n)$ time.

Even if the vertices are not neatly arranged, it is no big deal to handle; we can simply relabel all the vertices first and later the queries.

Now, let's think about a slightly more difficult scenario. Consider a graph that looks kind of like a tripod, like the following:

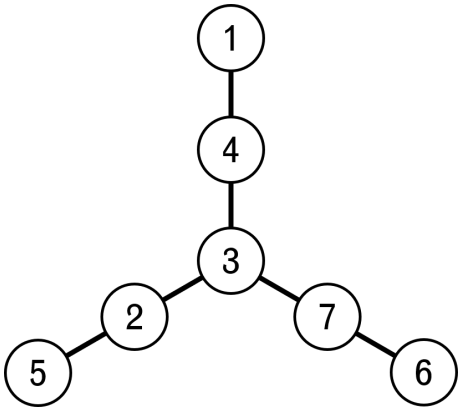

Figure 6: Tripod graph

How do we handle this? Well, we already know how to handle a line. How about we consider two lines: the line 1-4-3-2-5 and the line 7-6?

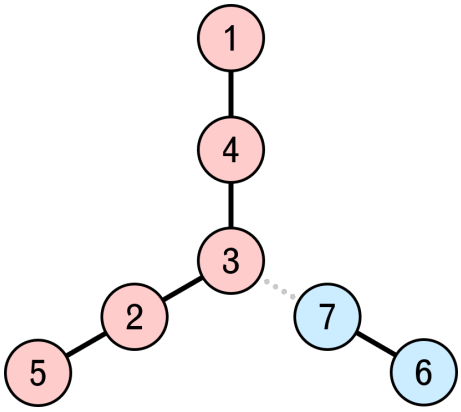

Figure 7: Tripod graph (decomposed)

Now, if it happens that u and v are both on the same line, then we're lucky; we can solve it quickly. But what if they are on different lines? Well, notice that, at least in a tripod graph, any path between two vertices can be decomposed into at most two paths that are both on lines. (We will discuss a systematic way of finding these "line paths"[11] later.)

In other words, any "tripod graph" query can be decomposed into two "segment tree" queries, and hence our runtime is still $\mathcal{O}(n + q \log n)$ on the tripod graph![12]

Let's see if we can extend the idea of "decomposing into line paths" to the general tree. As is always useful in these scenarios, we should consider rooting the tree (here, we standardize rooting it at vertex 1). Consider the following tree:

[11] paths where all vertices are part of the same line in the decomposition

[12] Assuming we can find the "line paths" quickly. For the tripod graph, at least, it's easy enough to do this using some case analysis.

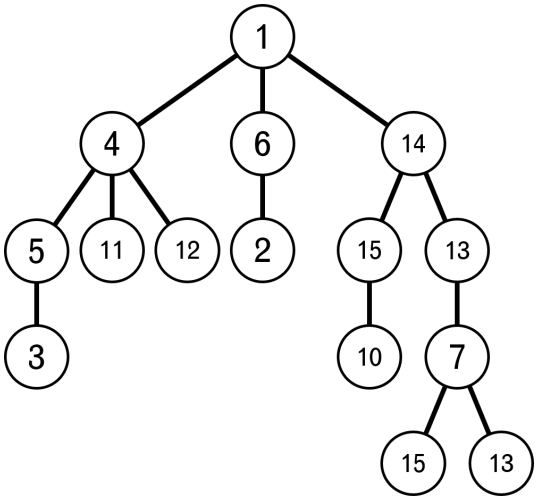

Figure 8: General tree

There are a few things we need to consider. First, how do we decompose the graph into line paths? Well, the number one thing we need to consider is that, between *any two* vertices, the number of times we need to transfer from one line path to another is small.

At this point, it's helpful to try it yourself. Can you come up with a good way to decompose a tree into lines? Remember that we want there to only be a few transfers from one line path to another. You can use the title of this section as a hint: "heavy-light decomposition".

Since we're minimizing the number of transfers, we want the paths to be long somehow, or at least split up the tree into multiple smaller parts. In other words, being *greedy* seems good. One systematic way to do it is to start from the root; this root has got to be part of some path. Connect this root to the child with the largest[13] subtree, with the rest of the children starting new paths. (In the event of a tie, continue with any of the largest children arbitrarily.) Then find the children's paths recursively.

If we do that here, we will arrive at the following line path decomposition:

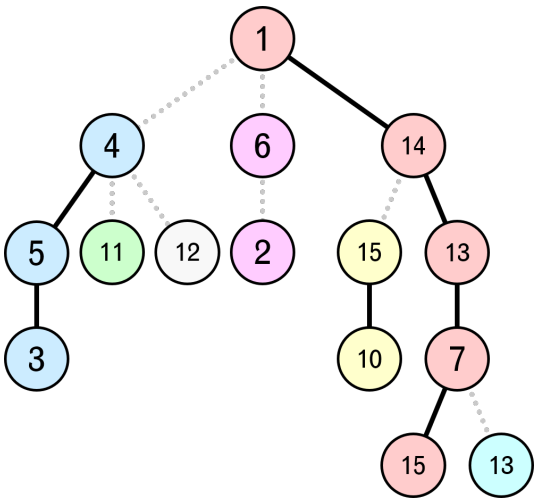

Figure 9: General tree (decomposed)

This decomposition can be done in $\mathcal{O}(n)$ with careful implementation. Now, let's think about the following: what is the largest number of times we will ever need to transfer from one

[13] in terms of the number of vertices

line path to another in a single graph query?

Well, suppose we started at some vertex u and want to go to v. Call $tr(u, v)$ the number of line transfers needed to go from u to v. It is clear that $tr(u, v) \leq tr(u, 1) + tr(1, v)$, considering that we rooted the tree at vertex 1.

We also have $tr(u, 1) + tr(1, v) \leq 2 \cdot \max(tr(1, u), tr(1, v))$.

So let's think about the largest possible value of $tr(1, x)$ for any x.

We are starting from vertex 1 and going downwards. Suppose our graph has n vertices and we need to transfer lines to go down. The number of transfers goes up by one, but notice that the number of vertices in the subtree we go down to is $< n/2$; this is because, any vertex is in the same line as its largest subtree, and if the subtree had at least half the vertices then surely it would be the largest child. Since we had to change lines, it isn't the largest child, it follows that it doesn't have at least half the vertices, so the number of vertices left is at least halved.

It immediately follows that $tr(1, x)$ is bounded by $\mathcal{O}(\log n)$, and hence so is $tr(u, v)$.

Assuming we can find the relevant line paths quickly, it therefore follows that any query on a general graph can be done in $\mathcal{O}(\log^2 n)$; there are $\mathcal{O}(\log n)$ relevant paths, each of which can be handled in $\mathcal{O}(\log n)$. So the only question left is how to find the relevant paths.

Well, finding the relevant paths is not really a big trouble, either!

Since we already know that there are only $\mathcal{O}(\log n)$ relevant paths, we should be content with being able to find all relevant paths in $\mathcal{O}(\log n)$. That is easily achievable with a straightforward algorithm.

First, we should do some preprocessing (after we have decomposed the graph into line paths); for every vertex, we need to determine who is the "head" of the line path this vertex is in; that is, the vertex closest to the root in the same line path. It may happen that the head is itself. So let $h(x)$ be the head of the line path that x is in.

Now, we can find the relevant paths in any query as follows; first, use any of the $\mathcal{O}(\log n)$ (or even $\mathcal{O}(1)$) lowest common ancestor algorithms to find the lowest common ancestor of u and v. Call this l.

Then start at u, and repeatedly do the following: first, check if u and l are in the same line path. If so, add the path u to l and stop; otherwise, add the path u to $h(u)$, then set u to the parent of $h(u)$ and then continue.

We can do the same for v; start at v and move upwards. Then, we will have all relevant paths, and we are almost done. There will always be at least one redundant line path which just looks like l to l (the lowest common ancestor to itself); throw one out. Then all the paths that are remaining are really the relevant line paths, and we can simply do the query separately on all the relevant line paths and then aggregate the result.

If you are using the standard binary lifting lowest common ancestor method, we can achieve a runtime of $\mathcal{O}(n \log n + q \log^2 n)$, which is generally good enough for these kinds of problems, and at least much better than $\mathcal{O}(nq)$.

In general, whenever you see a problem about querying or updating *paths* on a tree, which looks significantly easier if posed on an array, heavy-light decomposition may often be applicable.

6 Centroid Decomposition

Consider the following problem:

Problem 6.1. You have a tree with n vertices. Every edge has a non-negative integer length w_i. How many paths on the tree have a total length of exactly ℓ?

There is a straightforward $\mathcal{O}(n^2)$ algorithm—do a DFS or BFS from every vertex—but of course, we want a faster way to solve this problem.

As usual, try it out first before proceeding!

By now, you should have already learned the problem-solving technique of *trying easier cases first*. Solving it in an arbitrary tree is hard since we don't know what it looks like, so let's solve it first in trees which we know.

Basically the simplest tree is a line graph. In that case, the problem can be solved in $\mathcal{O}(n)$ time. It's probably too simple, so let's try the next one.

A slightly more difficult tree we can use is the tripod graph we encountered earlier while discussing heavy-light decomposition:

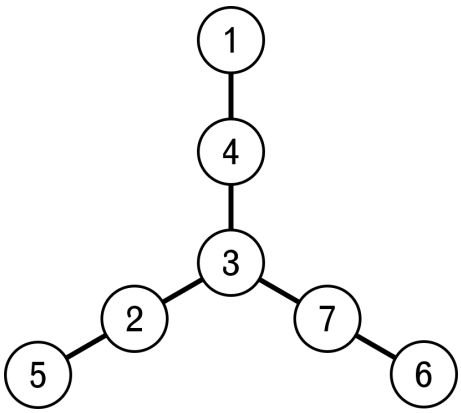

Figure 10: Tripod graph, again

Unlike the line graph, there's a very prominent vertex in this tree, the center, which is totally inviting us to *root the tree there*! So let's root the tree at the center.

We now have a rooted tree, and we can now try solving the problem recursively, with DP. First, let's recursively find the number of paths of length ℓ in each of the subtrees. Now we just need to count the number of paths that don't belong to any of the subtrees. Notice that these paths are exactly the paths that pass through the root.

Actually, we can be a bit more general now, and consider an *arbitrary* rooted tree instead of a tripod graph, since it's easy to count the number of paths through the root anyway!

To do so, we consider two kinds of paths that pass through the root:

- *The root is an endpoint.* In this case, the other endpoint must be a vertex with distance exactly ℓ away from the root. We can just compute the distances from the root with a single BFS or DFS in linear time.
- *The root is not an endpoint.* In this case, the two endpoints must belong to two different trees, and the root is an intermediate node.

  Consider some node, x. Let's count the number of such paths having x as one endpoint.

Its distance from the root—let's denote it by d_x—must be at most ℓ. The other endpoint—let's denote it by y— must be $d_y = \ell - d_x$ away from the root. Since the path passes through the root, y must also not be in the same subtree as x. Finally, these are sufficient—any y satisfying these is exactly ℓ away from x and the path passes through the root.

Luckily, just like in the previous case, we can easily find the number of nodes a distance of $\ell - d_x$ away from the root, after we've already computed all distances. This overcounts the paths though, since it includes the cases where the nodes are in the same subtree.

To fix this, we should just subtract them out. We can do this for each subtree one by one, so the overall cost is also linear.

This is now a complete solution, but how fast is it?

Let the children be $i_1, i_2, \ldots, i_k$. We're recursively solving the problem for each subtree, and then doing a linear-time procedure to count the number of paths passing through the root. Thus, if $T(r)$ is the running time for the subtree rooted at r, then we have the recurrence

$$T(r) = T(i_1) + T(i_2) + \ldots + T(i_k) + \mathcal{O}(\mathit{size}(r))$$

where $\mathit{size}(r)$ is the number of nodes in the subtree rooted at r. To solve this recurrence, first we note two things:

- The total number of nodes in each layer is $\mathcal{O}(n)$, so the total time for each recursion layer is $\mathcal{O}(n)$.
- The number of nodes decreases the deeper we go (since we're dropping at least one root every time), so the recursion depth is less than n.

Therefore, the running time is $\mathcal{O}(n^2)$. And we didn't seem to get anything better than brute force!

Let's look at what went wrong. First, we can express the running time as the tighter $\mathcal{O}(nh)$ where h is the maximum recursion depth of the algorithm. This is also basically the height of the tree. Writing it this way, we see that our solution is a bit faster if the tree has a small height. In the extreme case of a star tree, $h = 1$, so the solution is linear! And actually, we don't have to go that extreme. A perfect binary tree has $\mathcal{O}(\log n)$ height, so our solution runs in $\mathcal{O}(n \log n)$ for it. Finally, our solution also runs fast for random graphs since it turns out they have an expected height of $\mathcal{O}(\sqrt{n})$.

Unfortunately, we can't avoid the worst case of really long trees. In fact, our solution is $\mathcal{O}(n^2)$ for the tripod graph!

How do we improve this for the tripod graph? After rooting it at the center, we're left with three long paths. Our recursive solution will then take a really long time on these tall trees. This is kind of a waste, since line graphs are as simple as you get...we even derived a linear-time solution for them!

Luckily, we can improve the solution for the tripod graph by noticing that *we're actually free to reroot the subtrees*! We don't have to use the existing root as the root, since it isn't that special anyway. Instead, we probably want the center one.

If we root at the center node, then there will be two children that are both line graphs but of half the height. When we recurse on those line graphs, we again want the center node to be the root, halving the heights again. But since we're halving the height per recursion, it's clear that the number of recursions is only $\mathcal{O}(\log n)$, making the whole solution on the tripod graph $\mathcal{O}(n \log n)$!

So now we come to the wishful thinking part: It would be nice if there was a way to choose a vertex of any tree such that every subtree has "half the size" of the tree. And as it turns out, there is!

Theorem 6.1. For any unrooted tree with n nodes, there is a node such that if you root the tree at that node, then every proper subtree has size at most $n/2$.

The node satisfying Theorem 6.1 is called a **centroid** of the tree. Note that it's not the same as a *center* of the tree! Figure 11 illustrates the centroid of a tree.

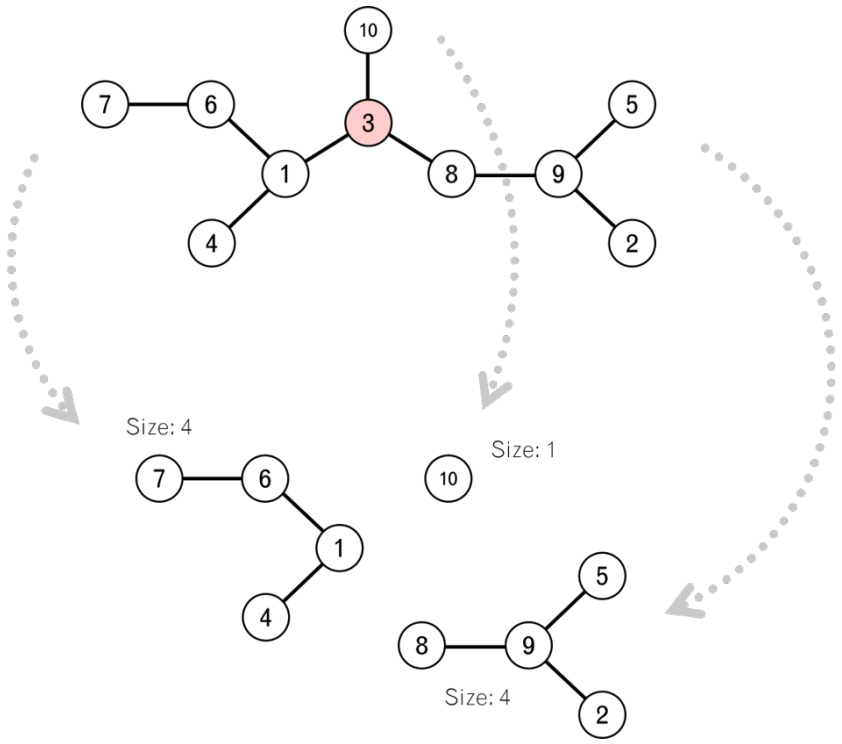

Figure 11: Centroid of a tree

Let's prove Theorem 6.1.

Proof. Note that in any rooted tree, there is at most one child with size more than half the size of the tree. Call such a child **heavy**. (What does this word "heavy" have to do with heavy-light decomposition?) Then a vertex is a centroid if and only if it has no heavy children.

Start at any vertex. If it doesn't have a heavy child, then it's already a centroid, and we're done. If not, go to the heavy child. Repeat the process until we either find a centroid, or we enter an infinite loop.

But we can prove that we won't enter an infinite loop as follows. Suppose otherwise, so there must be a sequence of nodes $x_1, x_2, \ldots, x_k$ such that x_{i+1} is the heavy child of x_i for $1 \leq i < k$, and x_1 is the heavy child of x_k. We clearly can't have $k = 1$. We also cannot have $k \geq 3$ since that would mean there's a cycle in our tree. Therefore, $k = 2$, and we have two adjacent nodes x_1 and x_2, each of which is the heavy child of the other.

But what are the subtree sizes exactly? Remove the edge (x_1, x_2) and we end up with two trees: T_1 (containing x_1) and T_2 (containing x_2). T_1 must have size $> n/2$ because x_1 is a heavy child of x_2, and T_2 must have size $> n/2$ because x_2 is a heavy child of x_1. But then, $T_1 \cup T_2$ must have size $> n$, which is a contradiction! Therefore, we won't run into an infinite loop, and there's at least one centroid.

Alternatively, instead of a proof by contradiction, we can also be more direct: notice that every time we move to a new vertex, the tree we "left behind" must have size $< n/2$ since we went to a heavy child. So it cannot be a heavy child, and we will never backtrack, which means

the algorithm will eventually halt (at a centroid). □

Notice that the proof is basically also the algorithm to find a centroid!

Exercise 6.2. Devise an $\mathcal{O}(n)$ algorithm to find a centroid.

We can now solve Problem 6.1! It's basically our earlier recursive solution, but with a twist: always choose a centroid as a root! The running time is clearly $\mathcal{O}(n \log n)$.

We can now make a bit more precise what "centroid decomposition" means. Notice that every vertex of the original tree would have been a centroid at exactly one recursion layer. If we draw the recursion layers as rows, then draw every node at the layer they were the centroid in, and connect it to its "parent" in the previous layer, then we form a tree, called the **centroid decomposition** of the tree.

Figure 12 shows a tree with the vertices colored according to layer. Can you draw the centroid decomposition as a tree?

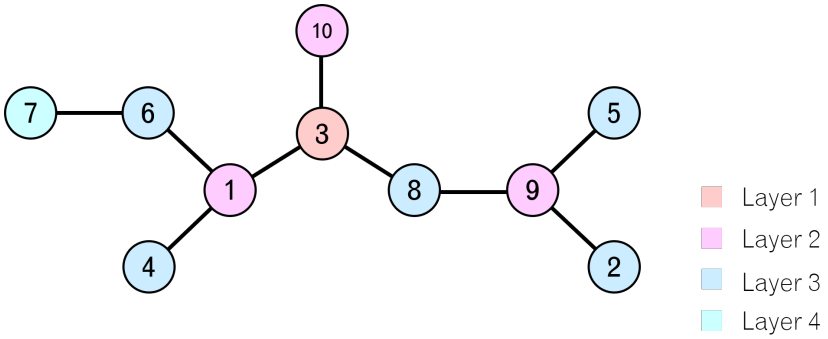

Figure 12: Centroid decomposition of a tree. Note that this image shows the edges of the original tree, not of the centroid decomposition "tree".

The centroid decomposition tree clearly has depth $\mathcal{O}(\log n)$, regardless of the original tree.

Centroid decomposition might be a good first line of offense when tackling problems regarding paths of a tree. It also has a few other applications which can be used to solve some rare query problems on trees; you are encouraged to read this Quora article for more.

7 Problems

As always, ask in Discord if anything is unclear!

7.1 Non-coding problems

No need to be overly formal in your answers!

N1 Prove that a tree has at most two centroids.

N2 Let $size(node)$ be the size of the subtree rooted at node $node$. Show that if $T(node) = \sum_{child} T(child) + \mathcal{O}(size(node))$, then $T(tree) = \mathcal{O}(size(node) \log size(node)))$.

N3 Let $height(node)$ be the height of the subtree rooted at node $node$. Show that if $T(node) = \sum_{child} T(child) + \mathcal{O}(height(node))$, then $T(tree) = \mathcal{O}(size(node))$.

Comparing this with the previous problem, this means that we don't get a log factor if we're only keeping $\mathcal{O}(height)$ amount of data per subtree instead of $\mathcal{O}(size)$.

N4 Compare the Heavy-Light Decomposition and the Long Path Decomposition. Is it also okay to use long path decomposition instead, whenever you need to use HLD? What would be the consequences?

7.2 Coding problems

C1 Use a combination of Euler tours and treaps to solve the following problem (dynamic forests) quickly:

> *There are n nodes, and initially zero edges. You need to process q queries:*
>
> - ***Link**. Given two nodes that belong to different connected components, add an edge between them.*
> - ***Cut**. Remove a given edge.*
> - ***Query**. Given two nodes, do they belong to the same connected component?*

C2 Generalize our reduction from RMQ to LCA. Solve the following problem in $\mathcal{O}((n + q) \log n)$ time by reducing it to LCA. (We can call it the *path minimum query problem* or the *bottleneck query problem* on a tree.)

> *Given a weighted tree with n nodes, answer q queries. In each query, you're given two nodes, and you need to find the minimum-weighted edge between those two nodes.*

C3 Solve the following problem in $\mathcal{O}((n + e + q) \log n)$ time.

> *Given a weighted undirected graph with n nodes and e edges, answer q queries. In each query, you're given two nodes, and you need to find the bottleneck between them. The bottleneck between two nodes i and j is defined as the maximum w such that there is a path from i to j all of whose edges have weights $\geq w$.*

Bonus: Solve it in $\mathcal{O}((n + e) \log n + q)$ time.

S1 Lomsat gelral: https://codeforces.com/problemset/problem/600/E

S2 Karen and Coffee: https://codeforces.com/problemset/problem/816/B
Try to avoid using complex data structures.

S3 Lolo Generoso: https://www.hackerrank.com/contests/noi-ph-2017/challenges/lolo-generoso

S4 Tourists: https://codeforces.com/problemset/problem/487/E
You will need some knowledge of the block-cut tree.

S5 Ciel and Commander: https://codeforces.com/problemset/problem/321/C

S6 Dick and a Box: https://www.hackerrank.com/contests/noi-ph-2017/challenges/dick-and-a-box
Note that an $\mathcal{O}(n)$ solution also exists, but you can pass it with $\mathcal{O}(n \log n)$.