# Alpine valley

**Spoiler**

**Abridged problem statement.** You are given a tree with $N$ vertices, with a special *exit vertex* $E$. Each edge has a positive weight. There are $S$ special vertices, we call them *shop vertices*. You are to answer $Q$ queries of the following form: suppose we temporarily erase edge $I$.

- Is it possible to reach vertex $E$ from vertex $R$ (without traversing $I$)?
- If not, how far is the closest shop vertex $R$ (without traversing $I$)?

**Subtask 1.** ($1 \leq N, Q \leq 100$; the graph looks like this: $\circ - \circ - \circ - \cdots - \circ$)

Removing an edge $(u, u+1)$ will split the graph to two parts: those with indices $1 \ldots u$ and those with indices $u+1 \ldots N$. It should be easy to check whether $E$ and $R$ are on the same "side". If they are, the answer is `escaped`. Otherwise, it is fairly straightforward to iterate over all shop vertices on the "side" of $R$ and calculate distances to find the closest one.

**Subtask 2.** ($1 \leq N, Q \leq 1000$)

The simplest solution (and typically, insufficient) to any problem is to simply do what the problem statement tells us to do, without giving it any more thought. This subtask can be solved by doing exactly that.

Suppose a query $(I, R)$ comes along. Then, we:

1. traverse the graph (pretending that the edge $I$ does not exist) to calculate, for each vertex $v$, the distance from $R$ to $v$;
2. if the distance from $R$ to $E$ is not $\infty$, announce that it is possible to reach $E$ from $R$;
3. otherwise, iterate over all shop vertices to find the one closest to $R$.

And this is done for all queries, separately.

Steps 2 and 3 should be straightforward to implement. There are many slightly different ways to do step 1. What makes it simpler is the fact that in a tree, there is exactly one path (that doesn't repeat vertices) between any two vertices. Thus the distance from $R$ to $v$ (for any $v$) is the length of that only path.

The idea is as follows. We know that the distance from $R$ to $R$ is 0. From this, we can calculate the distance from $R$ to its neighbours. Knowing the distance from $R$ to its neighbours, we can calculate the distance from $R$ to those neighbours' other neighbours. And so on. Formalizing and implementing this gives us something like the following:

**Algorithm 1:** Breadth-first search to answer queries

```
queue ← a first-in, first-out queue, initially empty
dist ← an array, initially dist[v] = ∞ for any v
add R to the back of queue
dist[R] ← 0
while queue is not empty do
    u ← first element of queue
    remove the first element from queue
    for each edge (u, v) from u do
        if we have not visited v and (u, v) ≠ I then
            add v to the back of queue
            dist[v] ← dist[u] + weight(u, v)
        end
    end
end
```

Here, `weight`$(u, v)$ denotes the weight of the edge between $u$ and $v$.

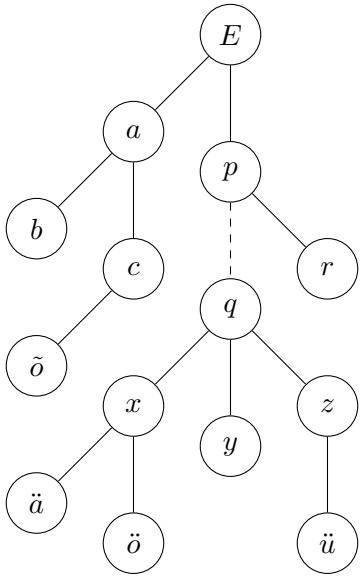
A practical note: the answer to the question can be up to $10^5 \cdot 10^9 = 10^{14}$. Thus, using `int` (in C++ and Java) will lead to overflow. Use `long` (in Java) or `long long` (in C++) instead.

Let's calculate the complexity of this solution. In algorithm 1, we visit each vertex and each edge at most once. Thus the complexity of step 1 is $\mathcal{O}(N)$. Steps 2 and 3 also take no more than this. As such, we take $\mathcal{O}(N)$ time for each query, for a total complexity of $\mathcal{O}(NQ)$.

If we only had one query, this would be optimal. But, we are neglecting the fact that all those queries take place on the same graph...

**Subtask 3.** ($1 \leq N, Q \leq 100\,000$; all vertices are shop vertices)

From this point on, we consider our graph as a *rooted tree*, rooted at the exit vertex $E$. It might look something like this:



Among the neighbours of a vertex, we distinguish between its *parent* and its *children*. For example, among the neighbours of $q$, the parent is $p$ and the children are $x$, $y$ and $z$.

Why is this representation convenient? Suppose we erased an edge, for example, the edge $(p, q)$, which is dashed in the picture above. Then we can verify that the answer is `escaped` for the vertices outside the subtree of $q$, and something else for the vertices inside the subtree of $q$. It is easy to confirm that this holds for any query $(I, R)$ — the answer is `escaped` if and only if $R$ does not lie in the "lower vertex" of the edge $I$.

In this subtask, this is enough! If we can't escape the valley, the vertex $R$ is itself a shop vertex. Therefore the answer to each query is either 0 or `escaped`. Now we only need a way to quickly tell if one vertex is in a subtree of another. There are various approaches, we describe a particularly elegant one.

**Algorithm 2:** A recursive function

```
Function dfs(u)
    print u
    for each child v of u do
        call dfs(v)
    end
    print u
end
```

Consider the function `dfs` in algorithm 2. It recursively explores the subtree of a vertex, printing the index of the current vertex upon "entering" and "exiting" that subtree. For example, the output of running `dfs`$(q)$ is

$$q \quad x \quad ä \quad ä \quad ö \quad ö \quad x \quad y \quad y \quad z \quad ü \quad ü \quad z \quad q$$

Let's look at the output of `dfs`$(E)$. It is easy to see that $u$ is in the subtree of $v$ if and only if the first occurrence of $u$ happens after the first occurrence of $v$ and the last occurrence of $u$ happens before the last occurrence of $v$. Indeed, if $u$ is truly in the subtree of $v$, then "entering" and "exiting" the vertex $u$ must occur while we are in the subtree of $v$ — between "entering" and "exiting" $v$.

After running `dfs`$(E)$ once, we can answer queries in $\mathcal{O}(1)$ time, using this criterion. Running `dfs`$(E)$ takes $\mathcal{O}(N)$ time, thus the time complexity is $\mathcal{O}(N + Q)$.

**Subtask 4.** $(1 \leq N, Q \leq 100\,000)$

We already know how to tell if the answer to a query is `escaped`. Thus in this section, we only focus on calculating the answer in the other case. Suppose we have a query $(I, R)$. Let $p$ be the "lower vertex" of $I$ and suppose $R$ is in the subtree of $p$. Then we need to calculate the closest shop vertex to $R$ within the subtree of $p$.

Let `lca`$(u, v)$ denote the lowest common ancestor of vertices $u$ and $v$ and `dist`$_E[u]$ be the distance from $E$ to $v$. We can calculate the array `dist`$_E[u]$ like we did in subtask 2. Notice that the distance between $u$ and $v$ is

$$\texttt{dist}_E[u] - \texttt{dist}_E(\texttt{lca}(u, v)) + \texttt{dist}_E[v] - \texttt{dist}_E(\texttt{lca}(u, v)).$$

Let $u$ be the closest shop vertex to $R$ within the subtree of $p$. Let's pretend we don't know where $u$ is, but somehow know $w = \texttt{lca}(R, u)$. Then we can calculate the answer to the query

as

$$\texttt{dist}_E[R] - 2\,\texttt{dist}_E[w] + \min_v \texttt{dist}_E[v],$$

where the minimum is taken over all shop vertices in the subtree of $w$. We define

$$\texttt{magic}[w] := -2\,\texttt{dist}_E[w] + \min_v \texttt{dist}_E[v].$$

Algorithm 3 provides a dynamic programming solution to calculate the array `magic`.

**Algorithm 3:** Calculating `magic`.

**Function** `buildMagic`($u$)
  **for** *each child* $v$ *of* $u$ **do**
    call `buildMagic` ($v$)
  **end**
  **if** $u$ *is a shop vertex* **then**
    $\texttt{magic}[u] \leftarrow \texttt{dist}_E[u]$
  **else**
    $\texttt{magic}[u] \leftarrow \infty$
  **end**
  **for** *each child* $v$ *of* $u$ **do**
    $\texttt{magic}[u] \leftarrow \min(\texttt{magic}[u], \texttt{magic}[v])$
  **end**
**end**
call `buildMagic` (E)
**for** *each vertex* $u$ **do**
  $\texttt{magic}[u] \leftarrow \texttt{magic}[u] - 2\,\texttt{dist}_E[u]$
**end**

Calculating $\texttt{dist}_E[R] + \texttt{magic}[w]$ gives the shortest path from $R$ to a shop, provided that $w$ is the "uppermost" vertex on the path. The "uppermost" vertex on the path will be on the path from $R$ to $p$. Thus, the answer to the query is

$$\texttt{dist}_E[R] + \min_w \texttt{magic}[w],$$

where the minimum is taken over the path from $R$ to $p$ (including both $R$ and $p$).

Now we only need a way to calculate $\min_w \texttt{magic}[w]$ quickly. There are many ways to take minimums over paths on a tree — we describe one which is called "binary lifting". We initialize two 2D arrays: `jumpVertex` and `jumpMagic`. We want $\texttt{jumpVertex}[u][k]$ to be the $2^k$-th ancestor $u$; that is — the result of moving $2^k$ steps towards $E$ from $u$. And $\texttt{jumpMagic}[u][k]$ should be the minimum of `magic` on the path between $u$ and the $\texttt{jumpVertex}[u][k]$ (including $u$, but excluding $\texttt{jumpVertex}[u][k]$). One can verify that the following relations hold:

$\texttt{jumpVertex}[u][0]$ is the parent of $u$;
$\texttt{jumpMagic}[u][0] = \texttt{magic}[u]$;
$\texttt{jumpVertex}[u][k] = \texttt{jumpVertex}[\texttt{jumpVertex}[u][k-1]][k-1]$;
$\texttt{jumpMagic}[u][k] = \min(\texttt{jumpMagic}[u][k-1], \texttt{jumpMagic}[\texttt{jumpVertex}[u][k-1]][k-1])$.

For example, the 8th ancestor of a vertex is the 4th ancestor of its 4th ancestor. We can use these equations to initialize the arrays `jumpVertex` and `jumpMagic`.

How can we use these arrays to take minimum of `magic` on the path from $R$ to $p$? We start at $R$, then use `jumpVertex` to jump up as far as possible without passing over $p$. The corresponding value of `jumpMagic` is the minimum of `magic` over all the vertices we skipped over. We keep jumping up until we reach $p$. Algorithm 4 provides the details.

**Algorithm 4:** Binary lifting

```
Function buildLifting(u, p) /* p is the parent of u                    */
    jumpVertex[u][0] ← p
    jumpMagic[u][0] ← magic[u]
    for k ← 1 to ⌈log2 N⌉ do
        /* jumping ⌈log2 N⌉ is enough to get us to the root            */
        jumpVertex[u][k] ← jumpVertex[jumpVertex[u][k − 1]][k − 1]
        jumpMagic[u][k] ← min(jumpMagic[u][k − 1], jumpMagic[jumpVertex[u][k − 1]][k − 1])
    end
    for each child v of u do
        call buildLifting (v, u)
    end
end
Function minPath(R, p)
    /* calculates the minimum of magic over the path from R to p.     */
    u ← R
    answer ← ∞
    for k ← ⌈log2 N⌉ to 0 do
        if jumpVertex[u][k] is in the subtree of p then
            answer ← min(answer, jumpMagic[u][k])
            u ← jumpVertex[u][k]
        end
    end
    answer ← min(answer, magic[p])
    return answer
end
```

To summarize:

1. we call `buildMagic`$(E)$ and other auxiliary pre-processing functions;
2. we call `buildLifting`$(E, E)$ to initialize the binary lifting tables;
3. for each query $(I, R)$, where $p$ is the "lower vertex" of $I$, answer the query as $\texttt{dist}_E[R] +$ `minPath`$(R, p)$ (or `escaped` if $R$ is not in the subtree of $p$).

Steps 1 should not take more than $\mathcal{O}(N)$ time. By analyzing the pseudocode we can see that step 2 takes $\mathcal{O}(N \log N)$ and step 3 takes up to $\mathcal{O}(\log N)$ for each query, i.e $\mathcal{O}(Q \log N)$ in total. Total complexity is $\mathcal{O}((N + Q) \log N)$.

**Credits**

- Task: TODO (Germany)
- Solutions and tests: Tähvend Uustalu, Andres Unt (Estonia)