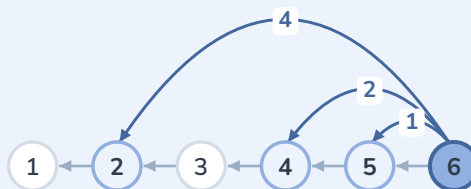


Binary lifting

Chapter 4.3 · Overall level: Advanced



In a family tree a million generations deep, *who was your ancestor exactly 700,000 generations ago?* Climbing one generation at a time is hopeless. Binary lifting lets you climb by 1, 2, 4, 8, ... generations at once, and reach any ancestor in about twenty jumps.

The same doubling table answers questions about lowest common ancestors, distances in trees and long runs of any “next” function, so it is one of the most reusable ideas in this part of the library.

4.3.1 Walking up one edge at a time

Chapter 4.1 stored a rooted tree by giving every vertex its *parent*, the next vertex on the way to the root. The root has no parent. Going up k edges from v means following parents k times; the vertex reached is the k -th ancestor of v .

A loop that follows one parent per step needs k steps for each question. On a tree shaped like a long chain, k can be about as large as n , so 200,000 questions on 200,000 vertices could cost 4×10^{10} steps. Most of those steps are repeated: every question that climbs through the same part of the chain walks it again.

4.3.2 Jumps whose lengths are powers of two

Binary lifting precomputes, for every vertex, where a jump of 1, 2, 4, 8, ... edges lands. Write

$\text{up}[j][v]$ = the vertex 2^j edges above v , or 0 if there is none

Level 0 is simply the parent: $\text{up}[0][v] = \text{parent}[v]$. Each higher level is built from the level below it, because a jump of 2^j edges is two jumps of 2^{j-1} edges, one after the other:

$\text{up}[j][v] = \text{up}[j - 1][\text{up}[j - 1][v]]$

Vertices are numbered from 1, so the number **0** is free to mean “no vertex”. This is the *zero sentinel*: the root’s parent is 0, and we also set $\text{up}[j][0] = 0$ for every level. A jump that would leave the tree lands on 0, and every further jump from 0 stays at 0, so “not enough ancestors” takes care of itself.

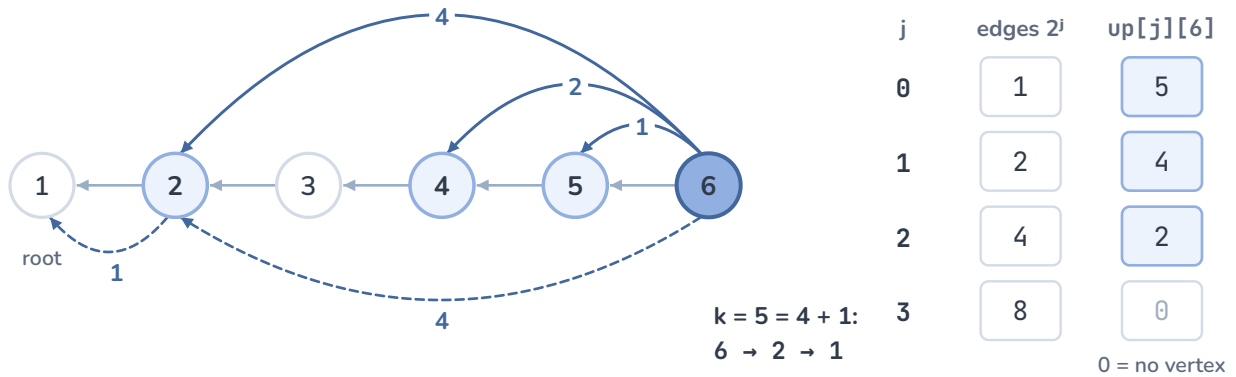


Figure 4.3.1 — Doubling stores repeated function application in powers of two. Each arc is labelled with its length in edges; the table pairs the level j with the actual distance 2^j . Level 3 would need 8 edges, so it holds the sentinel 0.

For the chain $1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5 \leftarrow 6$ with root 1, the whole table is small enough to write out. Column 0 is the sentinel.

	$v = 0$	1	2	3	4	5	6
$up[0][v]$ (1 edge)	0	0	1	2	3	4	5
$up[1][v]$ (2 edges)	0	0	0	1	2	3	4
$up[2][v]$ (4 edges)	0	0	0	0	0	1	2

Check one entry with the recurrence: $up[2][6] = up[1][up[1][6]] = up[1][4] = 2$. Four edges above 6 is indeed 2. And $up[1][2] = up[0][up[0][2]] = up[0][1] = 0$: two edges above 2 would be above the root, so the sentinel appears without any special case.

Why the table is right, and how to fill it

By induction on j . Level 0 is correct by definition. If level $j - 1$ is correct for every vertex, then $up[j - 1][v]$ is 2^{j-1} edges above v , and one more jump of the same length from there is 2^j edges above v . If either half leaves the tree, the result is 0, which is also correct.

The proof only uses the *whole* previous level, so fill the table one level at a time: for $j = 1, 2, \dots$, compute level j for every vertex. Do not fill it one vertex at a time. Parents are not always numbered smaller than their children, and level $j - 1$ of the parent may not exist yet.

Building L levels for n vertices takes $O(nL)$ time and memory.

4.3.3 Any distance from powers of two

Every whole number k is a sum of distinct powers of two: its binary digits. To jump k edges, look at the bits of k from the lowest upward, and whenever bit j is 1, make the jump $v \leftarrow \text{up}[j][v]$. For the chain and $k = 5 = 101$ in binary:

Bit j	Bit of k	Action	Vertex	Edges climbed so far
0	1	jump $2^0 = 1$: $\text{up}[0][6]$	5	1
1	0	skip	5	1
2	1	jump $2^2 = 4$: $\text{up}[2][5]$	1	5

The figure took the jumps in the order 4 then 1 and this table takes 1 then 4; both reach vertex 1. Order does not matter, because every jump applies the same parent step a number of times, and applying it a times and then b times is the same as b then a : $a + b$ times in all. The set bits add up to exactly k , so exactly k parent steps are applied.

If k is larger than the depth of v , some jump reaches 0 and the answer stays 0, meaning the ancestor does not exist. From the same vertex 6, $k = 6 = 2 + 4$ gives $6 \rightarrow 4 \rightarrow 0$.

How many levels?

The levels must cover the **largest** k a question can ask for, not just the number of vertices. If $k < 2^L$, bits 0 to $L - 1$ are enough. With $k \leq 200,000$ we need $L = 18$ levels, since $2^{17} = 131,072$ is too small and $2^{18} = 262,144$ is enough. With $k \leq 10^{12}$ we need 40 levels, bits 0 to 39: $2^{39} \approx 5.5 \times 10^{11} < 10^{12} < 2^{40} \approx 1.1 \times 10^{12}$.

Successor functions: cycles are fine

Nothing above used the fact that the parent links form a tree. Any function f that sends each vertex to exactly one next vertex can be doubled the same way, with $\text{up}[0][v] = f[v]$. Such a graph may contain cycles and self-loops, and every vertex always has a successor, so no sentinel is needed. For $f = [2, 3, 2, 4]$ on vertices 1 to 4:

	$v = 1$	2	3	4
$\text{up}[0][v]$	2	3	2	4
$\text{up}[1][v]$	3	2	3	4
$\text{up}[2][v]$	3	2	3	4

Applying f three times from 1 ($3 = 2 + 1$) gives $\text{up}[0][1] = 2$, then $\text{up}[1][2] = 2$. The path is 1, 2, 3, 2, so the answer 2 is right. Vertex 4 is a self-loop and stays 4 for any k .

Here k may be far larger than n because a walk can go round a cycle many times, so the number of levels comes from the bound on k alone: 40 levels for $k \leq 10^{12}$, even when $n = 4$.

Numbers larger than 32 bits

A value like 10^{12} does not fit in a 32-bit integer (limit about 2.1×10^9). Read k as `long long` in C++ and `Long` in Java. In JavaScript, ordinary numbers are exact up to 2^{53} , but the bit operators `&`, `>>` and `|` first cut their operands to 32 bits: `1e12 >> 1` evaluates to `-363689984`. Extract bits with arithmetic instead, `k % 2` and `Math.floor(k / 2)`, or use `BigInt`. Python integers have no size limit.

Pseudocode

k-th ancestor: $O(nL)$ to build, $O(L)$ per question

```
parent[root] ← 0; up[0] ← parent; up[j][0] ← 0 for every level j
for j from 1 to L - 1:                                # one whole level at a time
    for v from 1 to n:
        up[j][v] ← up[j - 1][ up[j - 1][v] ]
ancestor(v, k):
    j ← 0
    while k > 0 and v ≠ 0:
        if k mod 2 = 1: v ← up[j][v]
        k ← k div 2; j ← j + 1
    return v                                           # 0 means "no such ancestor"
```

4.3.4 Lowest common ancestors

In a rooted tree, the *lowest common ancestor* $\text{lca}(a, b)$ is the deepest vertex that is an ancestor of both a and b (a vertex counts as its own ancestor). The route between a and b climbs from a up to it and then goes down to b . Use the tree with edges 1–2, 1–3, 2–4, 2–5, 3–6 rooted at 1.

Rooting a tree given as an edge list

Trees often arrive as $n - 1$ undirected edges “ $u\ v$ ” with no parents given. Choose vertex 1 as the root and run a breadth-first search from it (Chapter 3.1): when a vertex w is first discovered from u , set $\text{parent}[w] = u$ and $\text{depth}[w] = \text{depth}[u] + 1$. The search uses a queue, not recursion, so a chain of 200,000 vertices causes no stack problem. Then build up from parent as before.

Step 1: bring both vertices to the same depth

If $\text{depth}[a] > \text{depth}[b]$, replace a by its ancestor $\text{depth}[a] - \text{depth}[b]$ edges up, using the jumps of Section 4.3.3. This does not change the answer: the LCA lies at or above the shallower vertex. For $a = 6, b = 2$ (depths 2 and 1), a becomes 3. If the two vertices are now equal, one was an ancestor of the other and that vertex is the LCA.

Step 2: jump both while their ancestors differ

Now $a \neq b$ at the same depth. For j from the highest level down to 0, look at $\text{up}[j][a]$ and $\text{up}[j][b]$. If they **differ**, the jump stays below the LCA, so make it with both vertices. If they are **equal**, the jump lands on the LCA or above it; it might overshoot, so refuse it. At the end, a and b are two different children of the LCA, and the answer is $\text{up}[0][a]$.

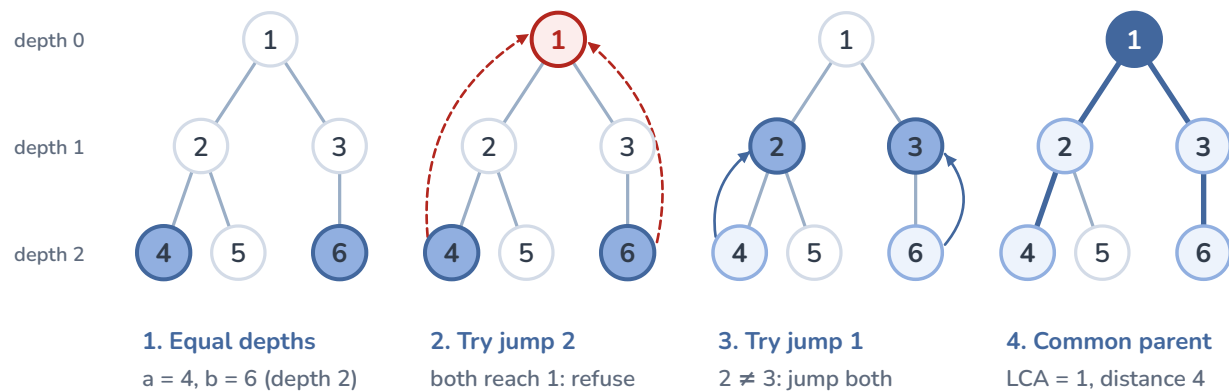


Figure 4.3.2 — After the largest safe paired jumps, the two vertices sit directly below their lowest common ancestor. Equal targets (outlined in red) may overshoot, so that jump is refused.

Why it works. Let the two vertices have depth d and let their LCA c have depth e . Climbing t edges from both gives the same vertex exactly when $t \geq d - e$, because below c the two routes to the root are separate. So the paired search is looking for the largest climb $T = d - e - 1$ with different results. Trying the powers of two from the top builds T bit by bit: a jump is taken exactly when adding it keeps the climb at most T . This is a binary search over the climb length. The invariant “the current a and b differ” holds after every step, and when all levels have been tried the climb equals T , leaving both vertices one edge below c .

Pseudocode

LCA: $O(L)$ per question after the $O(n L)$ table

```
lca(a, b):
    if depth[a] < depth[b]: swap a and b
    a ← ancestor(a, depth[a] - depth[b])    # step 1: equal depths
    if a = b: return a
    for j from L - 1 down to 0:              # step 2: paired jumps
        if up[j][a] ≠ up[j][b]:
            a ← up[j][a]; b ← up[j][b]
    return up[0][a]
```

The sentinel helps here too: a jump above the root gives 0 for both vertices, which counts as “equal”, so it is refused.

Distances in a tree

The route from a to b has $\text{depth}[a] - \text{depth}[c]$ edges up and $\text{depth}[b] - \text{depth}[c]$ edges down, where $c = \text{lca}(a, b)$. Adding them:

$$\text{distance}(a, b) = \text{depth}[a] + \text{depth}[b] - 2 \cdot \text{depth}[\text{lca}(a, b)]$$

Each depth counts the edges from the root. The part of the two root paths above the LCA is shared and is not on the route, and it was counted once inside each depth, so it is subtracted twice.

Pair	Depths	After step 1	LCA	Distance
4, 6	2, 2	4, 6 (no change)	1	$2 + 2 - 0 = 4$
4, 5	2, 2	4, 5 (no change)	2	$2 + 2 - 2 = 2$
6, 2	2, 1	3, 2	1	$2 + 1 - 0 = 3$
2, 2	1, 1	equal at once	2	$1 + 1 - 2 = 0$

4.3.5 Counting stops along a route

A related question asks for the vertex k edges along the route from a to b . Count carefully: k is an **edge offset** starting at 0, so offset 0 is a itself, offset 1 is the next vertex, and the last offset equals the distance. The route has one more vertex than it has edges.

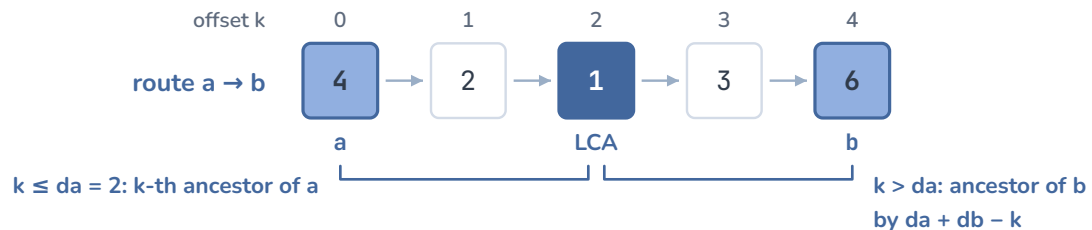


Figure 4.3.3 — Split the route at the LCA. Offsets on the rising part are ancestors of a ; offsets on the falling part are ancestors of b , counted from the other end.

With $da = \text{depth}[a] - \text{depth}[c]$ and $db = \text{depth}[b] - \text{depth}[c]$, the rising part holds offsets 0 to da and the falling part offsets da to $da + db$. A vertex on the falling part at offset k is $da + db - k$ edges from b , so it is an ancestor of b . For $a = 4$, $b = 6$: offset 2 is the 2nd ancestor of 4, which is 1; offset 3 is the $(4 - 3) = 1$ st ancestor of 6, which is 3. An offset larger than the distance has no vertex at all.

4.3.6 Code in four languages

Each program solves the first practice problem, **Kth ancestor**. The first line holds n and q ; the second line gives the parents of vertices 2 to n (it is empty when $n = 1$); then q lines “ v k ” follow. The tree is rooted at 1. For each question the program prints the k -th ancestor of v , or 0 if there are not enough ancestors.

Sample input	Sample output
6 3	1
1 1 2 2 3	0
4 2	6
4 3	
6 0	

Since $k \leq 200,000 < 2^{18}$, the programs build 18 levels. They use the zero sentinel, so no depths are needed: a question that climbs too far simply ends at 0. The bits of k are read with $k \% 2$ and division by 2, which stays correct if the bound on k is raised to 10^{12} and the level count to 40.

Python 3

```
import sys

LOG = 18                                # 2^18 > 200000, the largest k

def main():
    data = sys.stdin.buffer.read().split()
    n, q = int(data[0]), int(data[1])
    parent = [0, 0] + [int(x) for x in data[2:n + 1]]    # 0 means "no vertex"
    up = [parent]
    for j in range(1, LOG):
        prev = up[j - 1]
        up.append([prev[prev[v]] for v in range(n + 1)])
    out = []
    pos = n + 1
    for _ in range(q):
        v, k = int(data[pos]), int(data[pos + 1])
        pos += 2
        j = 0
        while k > 0 and v != 0:
            if k % 2 == 1:
                v = up[j][v]
            k //= 2
            j += 1
        out.append(v)
    print("\n".join(map(str, out)))

main()
```

C++17

```

#include <bits/stdc++.h>
using namespace std;

const int LOG = 18; // 2^18 > 200000, the largest k

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    // up[j][v] = vertex 2^j edges above v; 0 means "no vertex"
    vector<vector<int>> up(LOG, vector<int>(n + 1, 0));
    for (int v = 2; v <= n; v++) cin >> up[0][v];
    for (int j = 1; j < LOG; j++)
        for (int v = 1; v <= n; v++) up[j][v] = up[j - 1][up[j - 1][v]];
    while (q--) {
        int v;
        long long k;
        cin >> v >> k;
        for (int j = 0; k > 0 && v != 0; j++, k /= 2)
            if (k % 2 == 1) v = up[j][v];
        cout << v << '\n';
    }
    return 0;
}

```

JavaScript (Node)

```

const data = require("fs").readFileSync(0, "utf8")
    .trim().split(/\s+/).map(Number);
let pos = 0;
const n = data[pos++], q = data[pos++];
const LOG = 18; // 2^18 > 200000, the largest k
// up[j][v] = vertex 2^j edges above v; 0 means "no vertex"
const up = [new Int32Array(n + 1)];
for (let v = 2; v <= n; v++) up[0][v] = data[pos++];
for (let j = 1; j < LOG; j++) {
    const prev = up[j - 1], cur = new Int32Array(n + 1);
    for (let v = 1; v <= n; v++) cur[v] = prev[prev[v]];
    up.push(cur);
}
const out = [];
for (let t = 0; t < q; t++) {
    let v = data[pos++], k = data[pos++];
    for (let j = 0; k > 0 && v !== 0; j++) {
        if (k % 2 === 1) v = up[j][v];
        k = Math.floor(k / 2); // arithmetic, not >>, so k may exceed 32 bits
    }
    out.push(v);
}
console.log(out.join("\n"));

```

Java 17

```
import java.io.*;

public class Main {
    static final int LOG = 18; // 2^18 > 200000, the largest k
    static DataInputStream in = new DataInputStream(
        new BufferedInputStream(System.in, 1 << 16));

    static long next() throws IOException {
        int c = in.read();
        while (c <= ' ') c = in.read();
        long x = 0;
        while (c > ' ') {
            x = x * 10 + (c - '0');
            c = in.read();
        }
        return x;
    }

    public static void main(String[] args) throws IOException {
        int n = (int) next(), q = (int) next();
        // up[j][v] = vertex 2^j edges above v; 0 means "no vertex"
        int[][] up = new int[LOG][n + 1];
        for (int v = 2; v <= n; v++) up[0][v] = (int) next();
        for (int j = 1; j < LOG; j++)
            for (int v = 1; v <= n; v++) up[j][v] = up[j - 1][up[j - 1][v]];
        StringBuilder out = new StringBuilder();
        for (int t = 0; t < q; t++) {
            int v = (int) next();
            long k = next();
            for (int j = 0; k > 0 && v != 0; j++, k /= 2)
                if (k % 2 == 1) v = up[j][v];
            out.append(v).append('\n');
        }
        System.out.print(out);
    }
}
```

4.3.7 Cost

With L levels the table takes $O(nL)$ time and memory to build, and each ancestor, LCA or distance question takes $O(L)$. For 200,000 vertices and questions, $L = 18$ gives a table of 3.6 million entries and about 3.6 million jump steps for all the questions together. Vertex numbers fit in 32-bit integers; only k needs 64 bits when it can reach 10^{12} . With 40 levels the table grows to 8 million entries, about 32 MB at 4 bytes each, so compact arrays such as C++ `vector<int>` or JavaScript `Int32Array` matter there.

4.3.8 Common mistakes

- **Too few levels.** Levels must cover the largest k , not the number of vertices. A successor question with $n = 4$ and $k = 100$ needs 7 levels, and $k \leq 10^{12}$ needs 40, even though $\log_2 4 = 2$.
- **An undefined sentinel.** If the root's parent is -1 , the next jump reads $\text{up}[j][-1]$, outside the array. If the root is its own parent, a climb past the root returns 1 instead of "no ancestor". Use 0, and keep $\text{up}[j][0] = 0$.
- **Confusing the k -th vertex with k edges.** Offset 0 is the starting vertex. The route 4, 2, 1, 3, 6 has 4 edges and 5 vertices; offset 4 is 6, and offset 5 does not exist.
- **Filling the table vertex by vertex.** $\text{up}[j][v]$ needs level $j - 1$ of another vertex, which may have a larger number. Fill whole levels.
- **32-bit bit operators on large k .** In JavaScript, $k \gg 1$ and $k \& 1$ work on 32 bits only. In C++ and Java, read k into a 64-bit type.
- **Skipping step 1 of the LCA.** Paired jumps assume equal depths. Without step 1, the search can stop at two vertices whose parents differ.

4.3.9 When binary lifting is not the right tool

The table describes a fixed tree or a fixed successor function. If parents change between questions (edges are cut and re-linked), every level above the change may become wrong, and rebuilding costs $O(nL)$ each time; those dynamic trees need structures beyond this library. If there are only a few questions, a direct walk of k steps is simpler. For a single traversal of a whole tree, such as subtree sizes, the ordinary iterative traversal of Chapter 4.1 is enough. Questions that combine values along a path while values also change belong with the range structures of Chapter 4.2.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● Successor questions allow k up to 10^{12} . How many levels does the table need, and which bit positions do they cover?
2. ● Explain in one or two sentences why $\text{distance}(a, b) = \text{depth}[a] + \text{depth}[b] - 2 \cdot \text{depth}[\text{lca}(a, b)]$ is correct.
3. ● In the chain $1 \leftarrow 2 \leftarrow \dots \leftarrow 6$, which jumps answer $k = 3$ from vertex 6, and what do they reach? What is the answer for $k = 6$?
4. ●● For $f = [2, 3, 2, 4]$, find the vertex reached from 1 after applying f exactly 10^{12} times. Explain without building 40 levels by hand.
5. ●● In the tree 1–2, 1–3, 2–4, 2–5, 3–6 rooted at 1, find $\text{lca}(5, 6)$, the distance between 5 and 6, and the vertex at offset 1 on the route from 5 to 6.

Answers and explanations

Read these after attempting the exercises.

- 1. 40 levels**, bit positions 0 to 39. The largest k needs bit 39 ($2^{39} \approx 5.5 \times 10^{11} \leq 10^{12}$), and $2^{40} \approx 1.1 \times 10^{12}$ is already larger than any allowed k .
- Both depths count edges from the root. The root-to-LCA part is **shared**, counted once in each depth, and is not on the route, so it is subtracted twice. What remains is the climb from a to the LCA plus the descent to b .
- $3 = 2 + 1$: $\text{up}[0][6] = 5$, then $\text{up}[1][5] = 3$, so the answer is **3**. For $6 = 4 + 2$: $\text{up}[1][6] = 4$, then $\text{up}[2][4] = 0$. Vertex 6 has only 5 ancestors, so the answer is **0**.
- From 1 the walk is 1, 2, 3, 2, 3, ...: after $k \geq 1$ steps it is at 2 when k is odd and at 3 when k is even. 10^{12} is even, so the answer is **3**. The 40-level table finds the same result automatically.
- Depths are 2 and 2. A jump of 2 gives 1 and 1 (equal, refuse); a jump of 1 gives 2 and 3 (different, take it); the common parent is **1**. Distance = $2 + 2 - 0 = 4$. The route is 5, 2, 1, 3, 6, so offset 1 is **2**.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Kth ancestor

Apply

A tree is rooted at 1. For each query v k , move k parent edges upward from v . Print the reached vertex, or 0 if there are not enough ancestors. $k=0$ returns v .

Repeated successor

Connect

Each vertex has exactly one outgoing successor $f[v]$. For each v k , apply f exactly k times and print the result. Cycles and self-loops are allowed.

Tree route distance

Reason

Answer distances between pairs of vertices in an unweighted tree, measured in edges.

A stop along the route

Stretch

For each query a b k , follow the unique tree route from a toward b for k edges. Print that vertex, or 0 if k exceeds the route length. $k=0$ returns a .