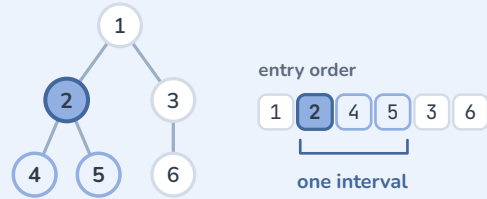


Trees

Chapter 4.1 · Overall level: Advanced



Folders inside folders, a tournament bracket, a family tree, a road network with no loops: all of them are trees. *Why can a computer answer a question about every branch of a tree at once, in a single pass?*

Because a tree has no cycles, every vertex hangs below exactly one other vertex once we pick a top. This chapter turns that observation into three tools: adding up values over whole branches, turning each branch into one stretch of an array, and finding the longest route in a tree with two sweeps.

4.1.1 What makes a graph a tree

A **tree** is a connected graph with no cycles. Three facts follow, and the algorithms in this chapter rely on all of them.

- **It has exactly $n - 1$ edges.** Start with the n vertices and no edges: n separate pieces. Add the edges one at a time. An edge that joins two vertices already in the same piece would close a cycle, so in a tree every edge joins two different pieces and reduces the number of pieces by exactly one. Ending with one connected piece takes exactly $n - 1$ edges. (This is the counting behind the disjoint-set structure of Chapter 3.2.)
- **Between any two vertices there is exactly one path.** There is at least one because the tree is connected. If there were two different paths, walking out along one and back along the other would contain a cycle.
- **Removing any edge disconnects it.** Otherwise the edge's two ends would still be joined by another path, giving a second path between them.

So a tree is the most economical way to connect n vertices. A graph with $n - 1$ edges that is not connected is a **forest**, several trees side by side; the practice problems guarantee a single tree, but real data often does not.

4.1.2 Rooting a tree: parent, children, depth, subtree

Pick one vertex as the **root**; the practice problems use vertex 1. Every other vertex v now has a unique path to the root, and the next vertex on that path is its **parent**. The vertices whose parent is v are its **children**. The **depth** of v is the number of edges from the root to v , and the **subtree** of v consists of v and every vertex below it, reached by going down through children again and again.

Throughout the chapter we use the tree with edges 1–2, 1–3, 2–4, 2–5 and 3–6, rooted at 1.

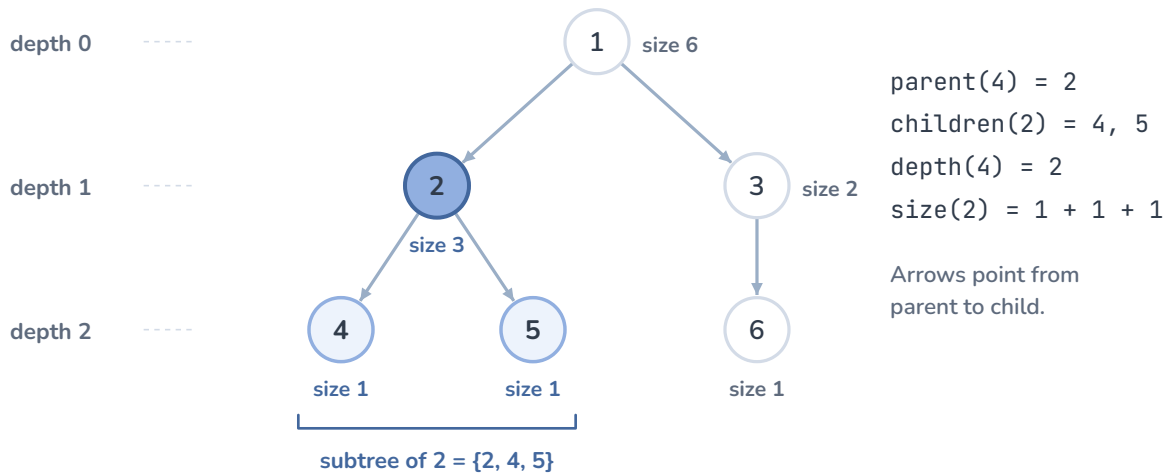


Figure 4.1.1 — A subtree contains a node and every descendant reached through its child edges. The subtree of 2 is {2, 4, 5}, so its size is 3.

The **subtree size** of v , the number of vertices in its subtree, follows a simple rule: the subtrees of different children never overlap (each vertex has one parent), so

$$\text{size}(v) = 1 + \text{size}(c_1) + \text{size}(c_2) + \dots \quad \text{over the children } c \text{ of } v$$

This is a dynamic program on the tree: the state is “the subtree of v ”, the transition adds the children, the base is a leaf (a vertex with no children) with size 1, and the **order** must handle every child before its parent. The next two sections produce such an order without recursion.

4.1.3 Traversal without recursion

The input lists edges in no particular direction, so we first discover the parent of every vertex. Start at the root and explore outwards. When we stand at v and look at a neighbour u , there are only two possibilities in a tree: u is the parent of v , which we came from, or u is a child. So the rule is simply **skip the parent edge**; no “visited” array is needed, because without cycles nothing else can lead back to an explored vertex.

A recursive depth-first search is the textbook way to explore, but a tree with 200,000 vertices can be a single long chain, and the recursion would then be 200,000 calls deep. Python stops at about 1000 levels by default, and C++, Java and JavaScript can overflow their call stacks too. An explicit **stack** does the same job with no depth limit:

Pseudocode

Entry order and parents: $O(n)$

```

parent[root] ← 0                (no parent)
stack ← [root]
order ← empty list
while stack is not empty:
    v ← pop the top of stack
    append v to order             (v is entered)
    for each neighbour u of v:
        if u ≠ parent[v]:
            parent[u] ← v
            push u onto stack

```

Here is the stack for the example tree. Children are pushed in decreasing order, so the smallest child is on top and is entered first; any pushing order would give a valid traversal.

Step	Pop (enter)	Push its children	Stack afterwards (top on the right)	order so far
1	1	3, 2	[3, 2]	1
2	2	5, 4	[3, 5, 4]	1, 2
3	4	—	[3, 5]	1, 2, 4
4	5	—	[3]	1, 2, 4, 5
5	3	6	[6]	1, 2, 4, 5, 3
6	6	—	[]	1, 2, 4, 5, 3, 6

A vertex is pushed only by its parent, after the parent was entered, so in order **every parent comes before all of its children**. Each vertex is pushed and popped once and each edge is looked at twice (once from each end), so the traversal takes $O(n)$ time.

4.1.4 Subtree sizes in reverse order

Reading order backwards gives exactly what the size recurrence needs: every child before its parent. Start every size at 1 (the vertex itself), then walk the order from the end and add each vertex's finished size to its parent.

Vertex taken (reverse order)	Its size is final	Add to parent	size[1..6] afterwards
start			1, 1, 1, 1, 1, 1
6	1	size[3] += 1	1, 1, 2, 1, 1, 1
3	2	size[1] += 2	3, 1, 2, 1, 1, 1
5	1	size[2] += 1	3, 2, 2, 1, 1, 1
4	1	size[2] += 1	3, 3, 2, 1, 1, 1
2	3	size[1] += 3	6, 3, 2, 1, 1, 1
1	6	(root: no parent)	6, 3, 2, 1, 1, 1

Why it is correct. When vertex v is taken, all of its children were taken earlier (they come after v in order), and each of them added its size to v . By induction on the reverse order, each child's size was already final when it was added, so $\text{size}[v]$ is now 1 plus the sizes of its children: the subtree rule. Every vertex adds itself exactly once, to its parent.

Pseudocode

Subtree sizes: $O(n)$ after the traversal

```

size[v] ← 1 for every vertex v
for v in order from the last to the second:    (the first is the root)
    size[parent[v]] ← size[parent[v]] + size[v]
output size[1], size[2], ..., size[n]
```

The same loop computes any quantity that combines over children: replace the initial 1 by a vertex's reward and you get **subtree sums**; replace the addition by $\max$ and you get the largest value in each subtree. Negative rewards change nothing in the argument.

4.1.5 Code in four languages

Each program solves the first practice problem, **Subtree sizes**. The first line holds n ; each of the next $n - 1$ lines holds an edge “ $u\ v$ ”. The output is the subtree size of every vertex, in vertex order, for the tree rooted at 1.

Sample input	Sample output
6 1 2 1 3 2 4 2 5 3 6	6 3 2 1 1 1

The programs push neighbours in input order, so their entry order may differ from the table above, but every order they produce has parents before children, which is all the sizes need. Java stores the adjacency lists in three plain arrays (head, next, to) to avoid creating 400,000 small objects.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    adj = [[] for _ in range(n + 1)]
    for k in range(n - 1):
        u, v = int(data[1 + 2 * k]), int(data[2 + 2 * k])
        adj[u].append(v)
        adj[v].append(u)
    parent = [0] * (n + 1)      # parent[1] = 0: the root has no parent
    order = []                 # entry order: parents before children
    stack = [1]
    while stack:
        v = stack.pop()
        order.append(v)
        for u in adj[v]:
            if u != parent[v]:
                parent[u] = v
                stack.append(u)
    size = [1] * (n + 1)
    for v in reversed(order):   # children before parents
        if v != 1:
            size[parent[v]] += size[v]
    print(" ".join(map(str, size[1:])))

main()
```

C++17

```

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<vector<int>> adj(n + 1);
    for (int k = 0; k < n - 1; k++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> parent(n + 1, 0), order, stk = {1};    // parent[1] = 0: root
    while (!stk.empty()) {
        int v = stk.back();
        stk.pop_back();
        order.push_back(v);                          // v is entered
        for (int u : adj[v])
            if (u != parent[v]) { parent[u] = v; stk.push_back(u); }
    }
    vector<int> size(n + 1, 1);
    for (int k = n - 1; k >= 1; k--)                  // reverse order, no root
        size[parent[order[k]]] += size[order[k]];
    for (int v = 1; v <= n; v++) cout << size[v] << (v == n ? '\n' : ' ');
    return 0;
}

```

JavaScript (Node)

```

const data = require("fs").readFileSync(0, "utf8")
    .trim().split(/\s+/).map(Number);
const n = data[0];
const adj = Array.from({ length: n + 1 }, () => []);
for (let k = 0; k < n - 1; k++) {
    const u = data[1 + 2 * k], v = data[2 + 2 * k];
    adj[u].push(v);
    adj[v].push(u);
}
const parent = new Int32Array(n + 1);    // parent[1] = 0: root
const order = [], stack = [1];
while (stack.length > 0) {
    const v = stack.pop();
    order.push(v);                        // v is entered
    for (const u of adj[v]) {
        if (u !== parent[v]) { parent[u] = v; stack.push(u); }
    }
}
const size = new Int32Array(n + 1).fill(1);
for (let k = n - 1; k >= 1; k--) size[parent[order[k]]] += size[order[k]];
console.log(Array.from(size.subarray(1)).join(" "));

```

Java 17

```

import java.io.*;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken();
        int n = (int) in.nval;
        // edges of v: e = head[v], then e = next[e] until -1; edge e leads to to[e]
        int[] head = new int[n + 1], next = new int[2 * n], to = new int[2 * n];
        Arrays.fill(head, -1);
        for (int e = 0; e < 2 * (n - 1); e += 2) {
            in.nextToken(); int u = (int) in.nval;
            in.nextToken(); int v = (int) in.nval;
            to[e] = v; next[e] = head[u]; head[u] = e;
            to[e + 1] = u; next[e + 1] = head[v]; head[v] = e + 1;
        }
        int[] parent = new int[n + 1], order = new int[n], stack = new int[n];
        int top = 0, cnt = 0;
        stack[top++] = 1; // parent[1] = 0: root
        while (top > 0) {
            int v = stack[--top];
            order[cnt++] = v; // v is entered
            for (int e = head[v]; e != -1; e = next[e])
                if (to[e] != parent[v]) { parent[to[e]] = v; stack[top++] = to[e]; }
        }
        int[] size = new int[n + 1];
        Arrays.fill(size, 1);
        for (int k = n - 1; k >= 1; k--) // reverse order, no root
            size[parent[order[k]]] += size[order[k]];
        StringBuilder out = new StringBuilder();
        for (int v = 1; v <= n; v++) out.append(size[v]).append(v < n ? " " : "\n");
        System.out.print(out);
    }
}

```

Sizes are at most 200,000, so 32-bit integers are enough here. Subtree *sums* of rewards up to 10^6 in absolute value can reach 2×10^{11} and need `long long` or `long`; a JavaScript Number holds them exactly.

4.1.6 Entry order: one subtree, one interval

The order in which vertices are entered (popped) has a second, very useful property: **the subtree of every vertex occupies one consecutive stretch** of it. Write $\text{tin}[v]$ for the position of v in the entry order, counting from 1. Then the subtree of v is exactly the positions

$$\text{tin}[v], \text{tin}[v] + 1, \dots, \text{tin}[v] + \text{size}[v] - 1$$

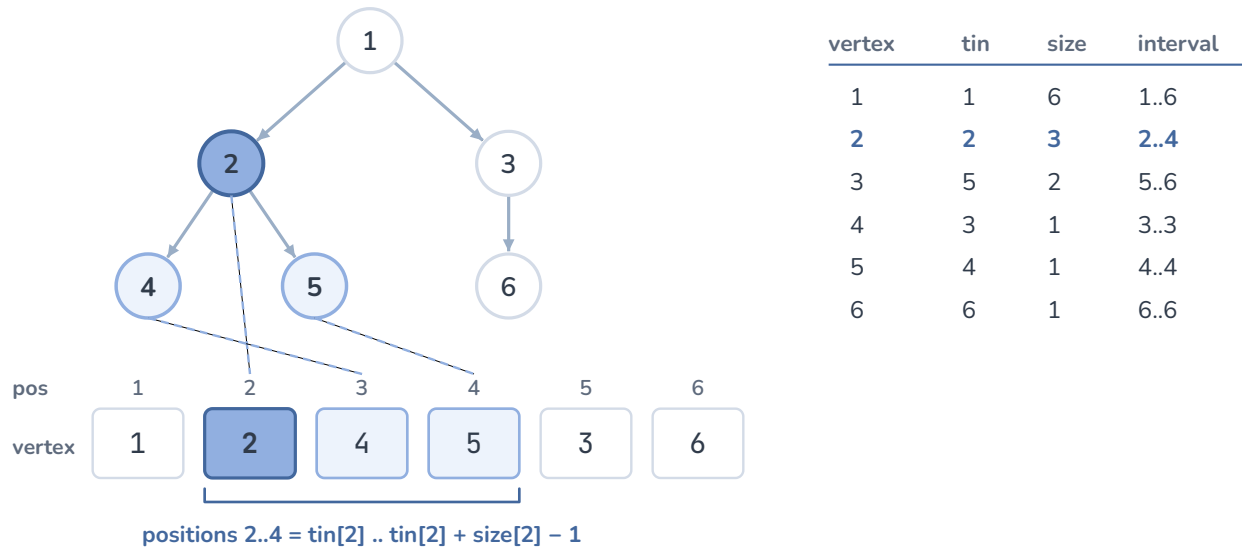


Figure 4.1.2 — Entering each node once maps every subtree to one interval. Vertex 2 is entered at position 2 and its subtree has 3 vertices, so it fills positions 2, 3 and 4.

Why it is true. Look at the moment v is popped, with some entries S lying below it in the stack. Its children are pushed on top of S . Popping a vertex only ever pushes that vertex's children, so until the stack shrinks back to S , every vertex popped is a descendant of v ; and every descendant is pushed (by its parent) before that happens. So right after v come all of its descendants and nothing else: $\text{size}[v]$ consecutive positions starting at $\text{tin}[v]$.

The convention matters. A breadth-first (queue) order, 1, 2, 3, 4, 5, 6 here, also puts parents first, which is enough for sizes, but it scatters the subtree $\{2, 4, 5\}$ over positions 2, 4 and 5. A depth-first *finishing* order (record a vertex when it is left), 4, 5, 2, 6, 3, 1, keeps subtrees together but puts each vertex *last* in its interval, so the formula above would give the wrong stretch. Fix one convention and use only its formula.

4.1.7 Subtree sums two ways

Give the vertices rewards $r = [5, -2, 4, 3, 1, 2]$ (for vertices 1 to 6). The total reward in every subtree can be computed in two ways, and both take $O(n)$.

Reverse aggregation, as for sizes but starting from $\text{sub}[v] = r[v]$, gives $\text{sub} = [13, 2, 6, 3, 1, 2]$.

Flatten, then prefix sums. Write the rewards in entry order: positions 1 to 6 hold vertices 1, 2, 4, 5, 3, 6, so the flat array is $[5, -2, 3, 1, 4, 2]$ and its prefix sums (Chapter 2.1) are $P = [0, 5, 3, 6, 7, 11, 13]$. Every subtree is an interval, so every subtree sum is one subtraction:

Vertex	Interval $\text{tin}.. \text{tin} + \text{size} - 1$	Sum from prefix array	Check
1	1..6	$P[6] - P[0] = 13 - 0 = 13$	all rewards
2	2..4	$P[4] - P[1] = 7 - 5 = 2$	$-2 + 3 + 1$
3	5..6	$P[6] - P[4] = 13 - 7 = 6$	$4 + 2$
4	3..3	$P[3] - P[2] = 6 - 3 = 3$	leaf

For fixed rewards the first way is simpler. The second matters later: it turns a subtree question into an interval question, and when rewards change, Chapter 4.2 maintains that interval with a Fenwick tree.

4.1.8 The diameter: two distance sweeps

The **distance** between two vertices is the number of edges on the unique path between them, and the **diameter** is the largest distance between any two vertices. Trying every pair is $O(n^2)$ or worse. A tree allows something much better:

1. From any vertex s , compute the distance to every vertex and pick a farthest vertex a .
2. From a , compute the distances again and pick a farthest vertex b .
3. The diameter is the distance from a to b , and the path $a...b$ is a longest path.

Distances come from the same stack traversal: when u is pushed from v , set $\text{dist}[u] = \text{dist}[v] + 1$. That is correct in a tree because the path through the parent is the only path. Breadth-first search (Chapter 3.1) works equally well.

On the example, the distances from 1 are 0, 1, 1, 2, 2, 2, so a can be 4 (vertices 5 and 6 tie; any of them works). From 4 the distances are 2, 1, 3, 0, 2, 4, so $b = 6$ and the diameter is 4, along 4–2–1–3–6. Note that the answer counts **edges**; the path has 5 vertices.

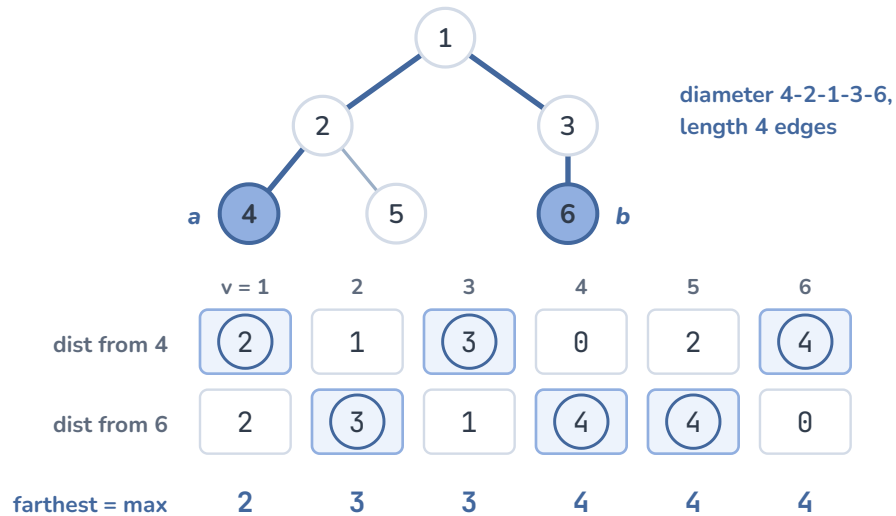


Figure 4.1.3 — A farthest endpoint of a diameter supplies each vertex's maximum distance. Distances from the two ends $a = 4$ and $b = 6$ are listed under each vertex; the circled value is the larger one.

Why the two sweeps work

Fix one diameter path P from u to v , of length D . For any vertex x , let p be the vertex of P closest to x ; the path from x to anything on P enters P at p .



Figure 4.1.4 — The picture behind both proofs. Branches leave the diameter path at p and q ; a branch can never be longer than the part of the path on either side of where it leaves.

Branch fact. If y joins P at q , then $d(q, y) \leq d(q, v)$ and $d(q, y) \leq d(q, u)$. Otherwise, say $d(q, y) > d(q, u)$; then the path from y through q to v would be longer than $d(u, q) + d(q, v) = D$, which is impossible.

Endpoint lemma. For all vertices x, y : $d(x, y) \leq \max(d(x, u), d(x, v))$. Indeed, if q lies between p and v (or $q = p$), then $d(x, y) \leq d(x, p) + d(p, q) + d(q, y) \leq d(x, p) + d(p, q) + d(q, v) = d(x, v)$; if q lies between u and p , the same steps give $d(x, u)$.

The first sweep finds a diameter end. Let a be farthest from the start s , let p be where s joins P and q where a joins it. Suppose q lies between p and v . Since a is farthest, $d(s, a) \geq d(s, v)$, that is $d(s, p) + d(p, q) + d(q, a) \geq d(s, p) + d(p, q) + d(q, v)$, so $d(q, a) \geq d(q, v)$. Then $d(u, a) = d(u, q) + d(q, a) \geq d(u, q) + d(q, v) = D$: the path from u to a is a longest path too, and a is a diameter end. If q lies on the other side, swap the roles of u and v . The second sweep from a then reaches distance exactly D .

4.1.9 The farthest vertex from every vertex

The **eccentricity** of a vertex is its largest distance to any vertex. Computing it from every vertex separately costs $O(n^2)$. The endpoint lemma gives it all at once: if a and b are the two ends of a diameter (from the two sweeps), then for every vertex x

$$\text{farthest}(x) = \max(d(x, a), d(x, b))$$

The lemma shows that no vertex is farther from x than the better of a and b , and both are actual vertices, so the maximum is attained. In Figure 4.1.3 the bottom row, 2, 3, 3, 4, 4, 4, is the eccentricity of vertices 1 to 6. It costs one extra distance sweep from b : three sweeps in total, $O(n)$.

4.1.10 Common mistakes

- **Treating a forest as a connected tree.** If the graph might have several components, one traversal from vertex 1 misses the others. Check that `order` contains all n vertices, or start a new traversal from every unvisited vertex.
- **Recursive DFS on 200,000 vertices.** A chain makes the recursion as deep as the tree. Use an explicit stack, as in this chapter; raising Python's recursion limit usually just turns the error into a crash.
- **Mixing vertex count and edge count.** The diameter of the example is 4 edges, along a path of 5 vertices. Read which one the problem asks for; a single vertex has diameter 0.
- **Using the wrong order for intervals.** Breadth-first order does not keep subtrees together, and finishing order puts each vertex last in its interval. Use the entry order with `tin[v] .. tin[v] + size[v] - 1`, or derive the formula for your own convention.

4.1.11 Where these methods stop

Everything here assumes a tree that does not change. With positive edge lengths the sweeps still work if distances add the length instead of 1, but on graphs with cycles the two-sweep trick fails; use the shortest-path methods of Chapter 3.3. Changing rewards need a Fenwick or segment tree over the entry order (Chapter 4.2), and questions about the path between two vertices, such as their lowest common ancestor, need binary lifting (Chapter 4.3).

Exercises

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

1. ● Why does a tree traversal skip the parent edge, and why is that enough to avoid revisiting vertices?
2. ● Is an arbitrary DFS finishing order the same as entry order for subtree intervals? Explain with the example tree.
3. ●● Root the tree with edges 1–2, 2–3, 2–4, 4–5, 4–6 at vertex 1. Push children in decreasing order and write the entry order, tin for every vertex and the subtree sizes. Which positions hold the subtree of 4?
4. ●● For the same tree, run the two sweeps from vertex 1. What are a, b and the diameter? Then give every vertex's farthest distance.

Answers and explanations

Read these after attempting the exercises.

1. To avoid immediately walking back. The parent is the only explored neighbour of v: a second one would give two paths from the root to v, which means a cycle.
2. No. Here the finishing order is 4, 5, 2, 6, 3, 1 and the entry order 1, 2, 4, 5, 3, 6. A vertex starts its interval in entry order but ends it in finishing order, so the formula depends on the exact convention.
3. Entry order 1, 2, 3, 4, 5, 6, so $\text{tin}[v] = v$. Sizes: 6, 5, 1, 3, 1, 1. The subtree of 4 is {4, 5, 6} at positions **4..6** = $\text{tin}[4] .. \text{tin}[4] + 3 - 1$.
4. From 1 the distances are 0, 1, 2, 2, 3, 3, so a = 5. From 5 they are 3, 2, 3, 1, 0, 2, so b = 1 (or 3) and the diameter is **3**, e.g. 5–4–2–1. The larger of the distances from 5 and from 1 gives the farthest distances **3, 2, 3, 2, 3, 3** for vertices 1 to 6.

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.

Subtree sizes

[Apply](#)

Root an undirected tree at vertex 1. For each vertex print the number of vertices in its subtree, including itself.

Subtree rewards

[Connect](#)

Each vertex has an integer reward. Root the tree at 1 and print the total reward in each vertex's subtree, including itself.

Tree diameter

[Reason](#)

Find the greatest distance between two vertices in an unweighted tree. Distance means number of edges.

Farthest from each vertex

[Stretch](#)

For every vertex of an unweighted tree, print its maximum distance to any vertex.