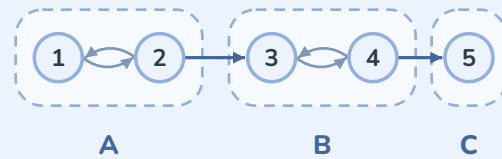


Strongly connected components

Chapter 4.4 · Overall level: Advanced



A city has some one-way streets. *From which places can you drive away and still find a legal route back?* Places that can all reach each other form a strongly connected component, and finding these groups turns a tangled directed graph into a clean one-way flow.

Once each group is squeezed into a single vertex, the graph has no cycles left, and the tools for acyclic graphs, such as topological order and dynamic programming, apply directly.

4.4.1 Mutual reachability in a directed graph

In a directed graph an edge $u \rightarrow v$ can be used only from u to v . Say u *reaches* v if some route follows edges from u to v ; a route of length zero is allowed, so every vertex reaches itself. Two vertices are **strongly connected** when each reaches the other.

The example for this chapter has five vertices and six edges:

$$1 \rightarrow 2, \quad 2 \rightarrow 1, \quad 2 \rightarrow 3, \quad 3 \rightarrow 4, \quad 4 \rightarrow 3, \quad 4 \rightarrow 5$$

Vertices 1 and 2 form a directed cycle, so they reach each other. Vertex 2 reaches 3 through the edge $2 \rightarrow 3$, but nothing leads from 3 back to 2: that edge is a *one-way bridge*. The same happens between 4 and 5. Vertex 1 reaches every vertex, but only 2 reaches 1.

Compare this with ordinary, undirected connectivity (Chapter 3.1). If we ignore directions, all five vertices lie in one piece; such a piece is called a *weakly connected* component. Strong connectivity is stricter and splits the same graph into three groups, the **strongly connected components** (SCCs): {1, 2}, {3, 4} and {5}.

Mutual reachability behaves like equality. Every vertex is related to itself; if u is related to v then v is related to u ; and if u and v reach each other and v and w reach each other, then joining routes shows that u and w do too. A relation with these three properties splits the vertices into separate groups, so every vertex belongs to **exactly one** SCC, even a vertex that is alone in its group, like 5.

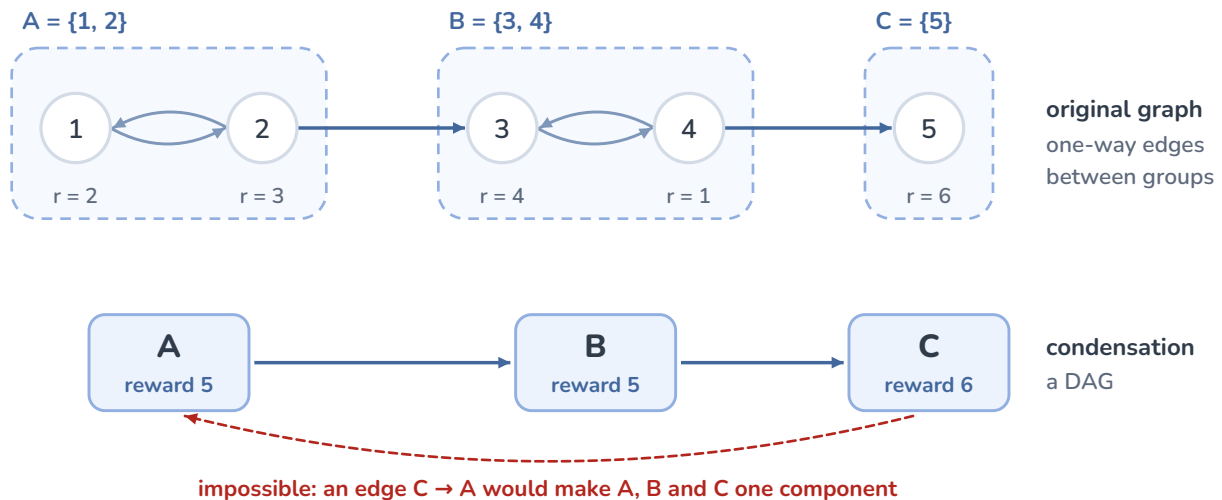


Figure 4.4.1 — A cycle between condensed components would make them one strongly connected component. Each component keeps the total reward of its vertices: $2 + 3 = 5$, $4 + 1 = 5$ and 6.

4.4.2 The condensation is acyclic

Replace every SCC by a single vertex, and keep an edge from component X to component Y whenever some original edge goes from a vertex of X to a vertex of $Y \neq X$. The result is the *condensation*. For the example it is the chain $A \rightarrow B \rightarrow C$.

The condensation never contains a cycle. Suppose components X and Y were on a cycle in the condensation. Then any vertex of X could reach any vertex of Y (walk inside X to the right exit, along the cycle, then inside Y), and the other way round. Every vertex of X would be strongly connected to every vertex of Y , so they would be one component, not two. In particular, two different SCCs can never have edges in both directions.

A directed graph with no cycles is a *DAG* (directed acyclic graph). So every directed graph is a DAG of SCCs, and each SCC is a group in which any vertex can get to any other. Many problems split along this line: inside a component, everything is reachable; between components, movement is one-way.

Two details matter when building the condensation. Edges inside one component, including self-loops, disappear. And several original edges can join the same pair of components, giving parallel condensed edges; you may remove the duplicates, or keep them as long as every later count treats them consistently.

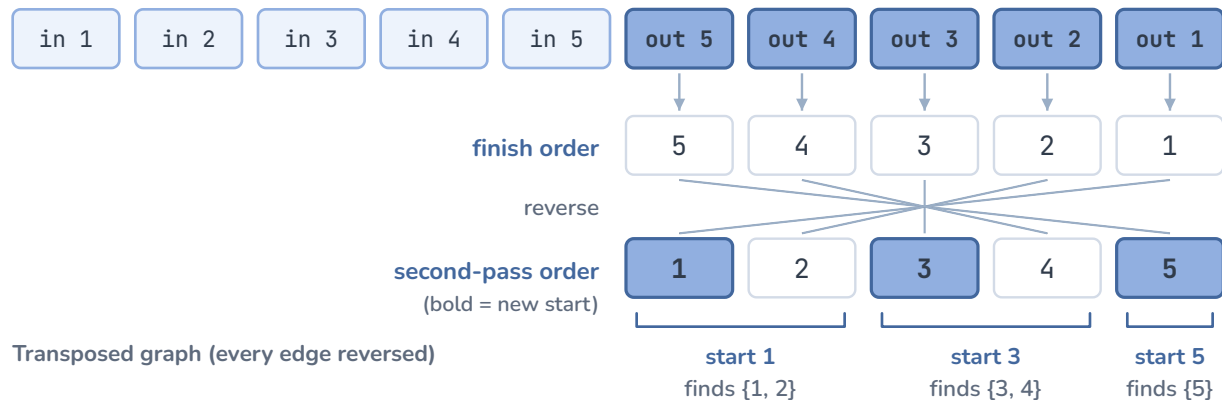
4.4.3 Finish order and the transposed graph

Finding the SCCs needs two ingredients. The first is a depth-first search (Chapter 3.1) that records two events for each vertex: its *entry*, when the search first arrives, and its *exit*, when every edge leaving it has been tried and the search goes back. A vertex that has exited is *finished*, and the order of exits is the **finish order**.

Run the search on the example from vertex 1, trying neighbours in increasing order. It enters 1, 2, 3, 4 and 5 along the edges $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$. Vertex 5 has no edges, so it exits first; then 4 (its other edge goes back to 3, already entered), 3, 2 and finally 1. The finish order is 5, 4, 3, 2, 1.

The second ingredient is the **transposed graph**: the same vertices with every edge reversed, so $u \rightarrow v$ becomes $v \rightarrow u$. Reversing edges does not change which pairs reach each other in both directions, so the transposed graph has exactly the same SCCs. Only the one-way edges between components change direction, and that is what keeps the second search from leaking out of a component.

Pass 1, original graph: entry (in) and exit (out) events, starting at 1



Transposed graph (every edge reversed)



Pass 2 follows reversed edges only.

From 2 it cannot reach 3, since $2 \rightarrow 3$ now points $3 \rightarrow 2$.

Figure 4.4.2 — The second pass starts from the latest finishing vertices, on the transposed graph. Each start collects exactly one component; vertices already collected are skipped.

4.4.4 Kosaraju's two-pass algorithm

Kosaraju's algorithm puts the two ingredients together:

1. Run depth-first searches on the original graph, starting a new search from every vertex not yet visited, and record the finish order.
2. Go through the vertices in **reverse** finish order. Each vertex that has no component yet starts a new component: search from it in the **transposed** graph, and every unassigned vertex reached joins that component.

In Figure 4.4.2 the second pass starts at 1 and collects only 2: the reversed edge $3 \rightarrow 2$ points into 2, not out of it. Next, 3 collects 4 through the reversed edge $3 \rightarrow 4$, and finally 5 is alone: three components.

Why it works

For a component X , let $f(X)$ be the latest finish time of its vertices. The key fact is:

Separation property. If the original graph has an edge from component X to a different component Y , then $f(X) > f(Y)$.

Two cases show it. **If the first search to touch X or Y enters X first**, at a vertex x , then everything in X and Y is reachable from x and still unvisited, so all of it is entered and finished before x itself exits. Thus x finishes last. **If it enters Y first**, that search can never reach X : a route from Y to X together with the edge from X to Y would make them one component. So all of Y finishes while X is still untouched, and X finishes later.

Now the second pass. Its first start is the vertex with the latest finish time overall, in some component X . In the transposed graph, an edge leaving X toward another component Z was originally an edge from Z to X , which by the property would give $f(Z) > f(X)$, impossible. So the search from the start cannot leave X , and it collects all of X because X is still strongly connected after reversal. Remove X and repeat the argument: each later start has the latest finish time among the unassigned vertices, any reversed edge to an unassigned component would contradict the property, and edges into assigned components lead to vertices that are skipped. By induction, **each second-pass search collects exactly one component**.

The same argument shows something useful for later: the second pass discovers components in an order where every condensed edge goes from an earlier component to a later one, which is a topological order of the condensation.

An explicit stack with exit events

A recursive depth-first search is the shortest way to write pass 1, but on a chain of 200,000 vertices it recurses 200,000 levels deep. Python stops at about 1,000 levels by default, and the other languages can run out of call stack too. So the programs in this chapter keep their own stack. The finish order needs a precise exit event, so each vertex also keeps a pointer to the next edge it has not tried yet:

- Look at the vertex u on top of the stack. If it still has an untried edge $u \rightarrow w$, advance the pointer; if w is unvisited, mark it and push it (the entry of w).
- If u has no untried edges left, pop it and append it to the finish order (the exit of u).

A vertex is pushed only once, when it is first marked, so the stack never holds more than n vertices. Pass 2 only needs to collect a set of vertices, not an order, so any stack or queue traversal works there.

Pseudocode

Kosaraju with explicit stacks: $O(n + m)$

```

pass 1 (original graph):
for s from 1 to n, if s is not seen:
    mark s seen; push s                # entry of s
    while the stack is not empty:
        u ← top of the stack
        if u has an untried edge u → w:
            move u's edge pointer forward
            if w is not seen: mark w seen; push w    # entry of w
        else:
            pop u; append u to order            # exit of u
pass 2 (transposed graph):
count ← 0
for s in order from last to first, if comp[s] = 0:
    count ← count + 1; comp[s] ← count; push s
    while the stack is not empty:
        pop u
        for each transposed edge u → w with comp[w] = 0:
            comp[w] ← count; push w
output count

```

Each vertex is pushed and popped once per pass and each edge is examined once per pass: $O(n + m)$ time and memory. Self-loops and parallel edges need no special handling.

4.4.5 Working on the condensation

With $\text{comp}[v]$ known for every vertex, two vertices are strongly connected exactly when their component numbers are equal. For the condensation, turn every edge $u \rightarrow v$ with $\text{comp}[u] \neq \text{comp}[v]$ into $\text{comp}[u] \rightarrow \text{comp}[v]$, in $O(n + m)$ time.

Topological order

A *topological order* of a DAG lists its vertices so that every edge goes from an earlier vertex to a later one. Kosaraju's second pass already numbers components in such an order. For any DAG, Kahn's algorithm finds one by repeatedly removing a vertex with no incoming edges:

```

indeg[X] ← number of condensed edges entering X, for every X
queue ← all X with indeg[X] = 0
while the queue is not empty:
    take X from the front; append X to topo
    for each condensed edge X → Y:
        indeg[Y] ← indeg[Y] - 1
        if indeg[Y] = 0: add Y to the queue

```

A vertex is queued only after all its incoming edges are removed, so its predecessors come earlier. Duplicate edges are harmless: each copy is counted once and removed once. For $A \rightarrow B \rightarrow C$ the order is A, B, C.

Source components

A component with no condensed edge entering it is a *source*. No vertex outside a source component can reach into it, so anything that must reach all vertices has to start inside every source.

Conversely, walking backwards along incoming edges in a DAG must stop, and only at a source, so every component is reachable from some source. In the example A is the only source.

Collecting rewards once

Give each vertex a nonnegative reward that can be collected only on the first visit, and allow a walk to repeat vertices. Inside one component a walk can visit every vertex and still leave through any exit, so entering a component is worth its **whole** reward, counted once. Between components the walk follows condensed edges, which never return. So the task becomes a longest-path DP on a DAG (Chapter 3.4): process components in topological order with

$$\text{best}[X] = \text{reward}[X] + \max(0, \text{best}[P] \text{ for every edge } P \rightarrow X)$$

Component	Vertices	Reward	best[X]
A	{1, 2}	2 + 3 = 5	5
B	{3, 4}	4 + 1 = 5	5 + 5 = 10
C	{5}	6	10 + 6 = 16

The largest best value, 16, is the answer; the walk 1, 2, 3, 4, 5 collects everything. A walk such as 1, 2, 1, 2, 3, 4, 3, 4, 5 is also allowed, and simply summing rewards along it would count four vertices twice.

4.4.6 Code in four languages

Each program solves the first practice problem, **Strong group count**. The first line holds n and m ; then m lines “ u v ” describe directed edges $u \rightarrow v$. Parallel edges and self-loops may occur. The program prints the number of strongly connected components.

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

The programs store the reversed edges while reading the input. Both passes call one function, `search`: an explicit-stack depth-first search that gives every unmarked vertex it reaches a label and records exit events. Pass 1 uses the label 1 and keeps the exits as the finish order; pass 2 uses the component number and ignores the exits.

Python 3

```
import sys

def search(g, s, mark, label, finished):
    # Iterative DFS from s: unmarked vertices reached get mark = label,
    # and each one is appended to finished at its exit event.
    mark[s] = label
    stack = [[s, 0]]                # vertex, index of its next edge
    while stack:
        top = stack[-1]
        u, k = top
        if k < len(g[u]):
            top[1] = k + 1
            w = g[u][k]
            if mark[w] == 0:
                mark[w] = label
                stack.append([w, 0])    # entry event for w
            else:
                stack.pop()
                finished.append(u)      # exit event: u is finished

def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    adj = [[] for _ in range(n + 1)]    # edges u -> v
    radj = [[] for _ in range(n + 1)]   # transposed edges v -> u
    for e in range(m):
        u, v = int(data[2 + 2 * e]), int(data[3 + 2 * e])
        adj[u].append(v)
        radj[v].append(u)
    seen, order = [0] * (n + 1), []
    for s in range(1, n + 1):           # pass 1: finish order
        if seen[s] == 0:
            search(adj, s, seen, 1, order)
    comp, count = [0] * (n + 1), 0     # comp 0 = not assigned yet
    for s in reversed(order):           # pass 2: transposed graph
        if comp[s] == 0:
            count += 1
            search(radj, s, comp, count, [])
    print(count)

main()
```

C++17

```

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

// Iterative DFS from s: unmarked vertices reached get mark = label,
// and each one is appended to finished at its exit event.
void search(const vector<vector<int>>& g, int s, vector<int>& mark, int label,
            vector<int>& finished) {
    vector<pair<int, int>> stk = {{s, 0}}; // vertex, index of its next edge
    mark[s] = label;
    while (!stk.empty()) {
        auto& [u, k] = stk.back();
        if (k < (int) g[u].size()) {
            int w = g[u][k++];
            if (mark[w] == 0) {
                mark[w] = label;
                stk.push_back({w, 0}); // entry event for w
            }
        } else {
            finished.push_back(u); // exit event: u is finished
            stk.pop_back();
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n + 1), radj(n + 1); // edges and transposed edges
    for (int e = 0; e < m; e++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        radj[v].push_back(u);
    }
    vector<int> seen(n + 1, 0), order, scratch;
    for (int s = 1; s <= n; s++) // pass 1: finish order
        if (seen[s] == 0) search(adj, s, seen, 1, order);
    vector<int> comp(n + 1, 0); // 0 = not assigned yet
    int count = 0;
    for (int k = n - 1; k >= 0; k--) // pass 2: transposed graph
        if (comp[order[k]] == 0) {
            count++;
            search(radj, order[k], comp, count, scratch);
        }
    cout << count << '\n';
    return 0;
}

```

Java 17

```

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

public class Main {
    static int[] stk, nxt, finished;    // explicit stack, next-edge pointers, exits
    static int done;

    // Iterative DFS: label every unmarked vertex reached; record exit events.
    static void search(List<Integer>[] g, int s, int[] mark, int label) {
        int top = 0;
        mark[s] = label;
        stk[top++] = s;
        while (top > 0) {
            int u = stk[top - 1];
            if (nxt[u] < g[u].size()) {
                int w = g[u].get(nxt[u]++);
                if (mark[w] == 0) {
                    mark[w] = label;
                    stk[top++] = w;                // entry event for w
                }
            } else {
                finished[done++] = u;            // exit event: u is finished
                top--;
            }
        }
    }

    @SuppressWarnings("unchecked")
    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;
        in.nextToken(); int m = (int) in.nval;
        List<Integer>[] adj = new List[n + 1], radj = new List[n + 1];
        Arrays.setAll(adj, v -> new ArrayList<>());
        Arrays.setAll(radj, v -> new ArrayList<>());    // transposed edges
        for (int e = 0; e < m; e++) {
            in.nextToken(); int u = (int) in.nval;
            in.nextToken(); int v = (int) in.nval;
            adj[u].add(v);
            radj[v].add(u);
        }
        stk = new int[n];                // a vertex enters the stack at most once
        nxt = new int[n + 1];
        finished = new int[2 * n];        // pass 1 exits, then pass 2 exits
        int[] seen = new int[n + 1], comp = new int[n + 1];    // comp 0: none yet
        for (int s = 1; s <= n; s++)    // pass 1: finish order
            if (seen[s] == 0) search(adj, s, seen, 1);
        Arrays.fill(nxt, 0);            // fresh edge pointers
        int count = 0;
        for (int k = n - 1; k >= 0; k--)    // pass 2: transposed graph
            if (comp[finished[k]] == 0) search(radj, finished[k], comp, ++count);
        System.out.println(count);
    }
}

```

JavaScript (Node)

```

const data = require("fs").readFileSync(0, "utf8")
  .trim().split(/\s+/).map(Number);
const n = data[0], m = data[1];
const adj = Array.from({ length: n + 1 }, () => []);
const radj = Array.from({ length: n + 1 }, () => []); // transposed edges
for (let e = 0; e < m; e++) {
  const u = data[2 + 2 * e], v = data[3 + 2 * e];
  adj[u].push(v);
  radj[v].push(u);
}
const stk = new Int32Array(n), edge = new Int32Array(n); // explicit stack

// Iterative DFS from s: unmarked vertices reached get mark = label,
// and each one is appended to finished at its exit event.
function search(g, s, mark, label, finished) {
  let top = 0;
  mark[s] = label;
  stk[top] = s;
  edge[top++] = 0;
  while (top > 0) {
    const u = stk[top - 1];
    if (edge[top - 1] < g[u].length) {
      const w = g[u][edge[top - 1]++];
      if (mark[w] === 0) {
        mark[w] = label;
        stk[top] = w;
        edge[top++] = 0; // entry event for w
      }
    } else {
      finished.push(u); // exit event: u is finished
      top--;
    }
  }
}

const seen = new Int32Array(n + 1), order = [];
for (let s = 1; s <= n; s++) { // pass 1: finish order
  if (seen[s] === 0) search(adj, s, seen, 1, order);
}
const comp = new Int32Array(n + 1); // 0 = not assigned yet
let count = 0;
for (let k = n - 1; k >= 0; k--) { // pass 2: transposed graph
  if (comp[order[k]] === 0) search(radj, order[k], comp, ++count, []);
}
console.log(count);

```

4.4.7 Cost

Both passes are linear: $O(n + m)$ time and $O(n + m)$ memory. Every number in the programs is a vertex number or an index, so 32-bit integers suffice. Reward totals are different: 200,000 rewards of up to 10^6 sum to 2×10^{11} , which needs `long` in C++ and `long` in Java.

4.4.8 Common mistakes

- **Using visit order instead of finish order.** For the single edge $1 \rightarrow 2$, the search enters 1 then 2, but finishes 2 then 1. Starting pass 2 at the last-entered vertex 2 follows the reversed edge $2 \rightarrow 1$ and wrongly puts 1 and 2 together. In this chapter's example the two orders happen to agree, so test with a graph where they differ.
- **Forgetting to transpose.** Searching the original graph in pass 2 from vertex 1 reaches all five vertices and reports one component.
- **Collecting a reward twice.** On a cycle a walk passes the same vertex many times. Sum rewards per component once, then work on the acyclic condensation.
- **Recursion on deep graphs.** A recursive search fails on a long chain. Use an explicit stack with an edge pointer.
- **Recording exit too early.** A vertex exits only after all its edges have been tried, not when it is popped for the first time or when its neighbours are pushed.
- **Comparing component numbers between programs.** Numbers are labels; only equality between them means something.

4.4.9 Limits of the method

SCCs describe a fixed graph. If edges are added or removed between questions, the components can merge or split, and the two passes must be repeated. For undirected graphs, components are simpler: use breadth-first search (Chapter 3.1) or disjoint sets (Chapter 3.2). Equal component numbers answer “do a and b reach each other?”, but one-way questions such as “does a reach b ?” between different components are not answered by the numbers alone. Tarjan's algorithm finds the same components in a single pass with low-link values; it is a good next step but is not needed here.

Exercises

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

1. ● Can two different SCCs have edges in both directions between them? Explain.
2. ● Must the component numbers be the same in every correct implementation?
3. ● For the edges $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 1$, $3 \rightarrow 4$, list the strongly connected components and the weakly connected components.
4. ●● Run pass 1 on the graph of Exercise 3 from vertex 1, neighbours in increasing order. List the entry and exit events, the finish order, and the components found by pass 2.
5. ●● In the chapter example with rewards 2, 3, 4, 1, 6, replace the edge $4 \rightarrow 5$ by $1 \rightarrow 5$. Draw the new condensation and find the largest reward a walk can collect.

Answers and explanations

Read these after attempting the exercises.

- 1. No.** An edge from X to Y and one from Y to X would let every vertex of each reach every vertex of the other, so all of them would be mutually reachable and form one component.
- 2. No.** The numbers depend on the order of starts and neighbours. Only the grouping is fixed, which is why answers should use counts, equality tests or sums, not the labels themselves.
- 3. SCCs: {1, 2, 3}** (a directed cycle) and **{4}**. Ignoring directions, everything is joined, so there is **one** weakly connected component.
- 4.** Events: in 1, in 2, in 3, in 4, out 4, out 3, out 2, out 1, so the finish order is 4, 3, 2, 1. Pass 2 takes 1 first; in the transposed graph $1 \rightarrow 3$, $3 \rightarrow 2$ and $2 \rightarrow 1$ lead around the cycle, but the reversed edge $4 \rightarrow 3$ cannot be followed from 3. Components: **{1, 2, 3}**, then **{4}**.
- 5.** The SCCs are unchanged: $A = \{1, 2\}$, $B = \{3, 4\}$, $C = \{5\}$, but now $A \rightarrow B$ and $A \rightarrow C$, and $B \rightarrow C$ is gone. $\text{best}[A] = 5$, $\text{best}[B] = 10$, $\text{best}[C] = 5 + 6 = 11$. A walk can no longer visit both B and C , so the answer is **11**.

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.

Strong group count

Apply

Count strongly connected components of a directed graph. Two vertices share a component iff each is reachable from the other.

Mutual reachability desk

Connect

For each pair a, b , decide whether there is a directed route from a to b and also a directed route from b to a . Length-zero routes are allowed.

Broadcast starters

Reason

Choose the smallest number of starting vertices so that every vertex is reachable from at least one chosen vertex in a directed graph. Print that minimum number.

Collect once on a walk

Stretch

Each vertex has a nonnegative reward. A directed walk may start and end anywhere and revisit vertices or edges. Each vertex reward is collected only on its first visit. Find the largest collectible reward.