```
/*

A graph consists of a set of objects (a.k.a vertices, or nodes)
and a set of
connections (a.k.a. edges) between pairs of said objects. A graph
may be stored
as an adjacency list, which is a space efficient representation
that is also
time-efficient for traversals.

The following class implements a simple graph using adjacency
lists, along with
depth-first search and a few other applications. The constructor
takes a Boolean
argument which specifies whether the instance is a directed or
undirected graph.
The nodes of the graph are identified by integers indices numbered
consecutively
starting from 0. The total number of nodes will automatically
increase based on
the maximum node index passed to add_edge() so far.

Time Complexity:
- O(1) amortized per call to add_edge(), or O(max(n, m)) for n
calls where the
  maximum node index passed as an argument is m.
- O(max(n, m)) per call for dfs(), has_cycle(), is_tree(), or
is_dag(), where n
  is the number of nodes and and m is the number of edges.
- O(1) per call to all other public member functions.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of
nodes and m
  is the number of edges.
- O(n) auxiliary stack space for dfs(), has_cycle(), is_tree(),
and is_dag().
- O(1) auxiliary for all other public member functions.

*/

#include <algorithm>
#include <vector>

class graph {
  std::vector<std::vector<int> > adj;
  bool directed;

  template<class ReportFunction>
  void dfs(int n, std::vector<bool> &visit, ReportFunction f)
const {
```

```
    f(n);
    visit[n] = true;
    std::vector<int>::const_iterator it;
    for (it = adj[n].begin(); it != adj[n].end(); ++it) {
      if (!visit[*it]) {
        dfs(*it, visit, f);
      }
    }
  }

  bool has_cycle(int n, int prev, std::vector<bool> &visit,
                 std::vector<bool> &onstack) const {
    visit[n] = true;
    onstack[n] = true;
    std::vector<int>::const_iterator it;
    for (it = adj[n].begin(); it != adj[n].end(); ++it) {
      if (directed && onstack[*it]) {
        return true;
      }
      if (!directed && visit[*it] && *it != prev) {
        return true;
      }
      if (!visit[*it] && has_cycle(*it, n, visit, onstack)) {
        return true;
      }
    }
    onstack[n] = false;
    return false;
  }

 public:
  graph(bool directed = true) : directed(directed) {}

  int nodes() const {
    return (int)adj.size();
  }

  std::vector<int>& operator[](int n) {
    return adj[n];
  }

  void add_edge(int u, int v) {
    if (u >= (int)adj.size() || v >= (int)adj.size()) {
      adj.resize(std::max(u, v) + 1);
    }
    adj[u].push_back(v);
    if (!directed) {
      adj[v].push_back(u);
    }
  }
```

```
  bool is_directed() const {
    return directed;
  }

  bool has_cycle() const {
    std::vector<bool> visit(adj.size(), false);
    std::vector<bool> onstack(adj.size(), false);
    for (int i = 0; i < (int)adj.size(); i++) {
      if (!visit[i] && has_cycle(i, -1, visit, onstack)) {
        return true;
      }
    }
    return false;
  }

  bool is_tree() const {
    return !directed && !has_cycle();
  }

  bool is_dag() const {
    return directed && !has_cycle();
  }

  template<class ReportFunction>
  void dfs(int start, ReportFunction f) const {
    std::vector<bool> visit(adj.size(), false);
    dfs(start, visit, f);
  }
};

/*** Example Usage and Output:

DFS order: 0 1 2 3 4 5 6 7 8 9 10 11

***/

#include <cassert>
#include <iostream>
using namespace std;

void print(int n) {
  cout << n << " ";
}

int main() {
  {
    graph g;
    g.add_edge(0, 1);
    g.add_edge(0, 6);
    g.add_edge(0, 7);
    g.add_edge(1, 2);
```

```
    g.add_edge(1, 5);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(7, 8);
    g.add_edge(7, 11);
    g.add_edge(8, 9);
    g.add_edge(8, 10);
    cout << "DFS order: ";
    g.dfs(0, print);
    cout << endl;
    assert(g[0].size() == 3);
    assert(g.is_dag());
    assert(!g.has_cycle());
  }
  {
    graph tree(false);
    tree.add_edge(0, 1);
    tree.add_edge(0, 2);
    tree.add_edge(1, 3);
    tree.add_edge(1, 4);
    assert(tree.is_tree());
    assert(!tree.is_dag());
    tree.add_edge(2, 3);
    assert(!tree.is_tree());
  }
  return 0;
}


/*

Given a directed acyclic graph, find one of possibly many
orderings of the nodes
such that for every edge from node u to v, u comes before v in the
ordering.
Depth-first search is used to traverse all nodes in post-order.

toposort(nodes) takes a directed graph stored as a global
adjacency list with
nodes indexed from 0 to (nodes - 1) and assigns a valid
topological ordering to
the global result vector. An error is thrown if the graph contains
a cycle.

Time Complexity:
- O(max(n, m)) per call to toposort(), where n is the number of
nodes and m is
  the number of edges.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of
```

```
nodes and m
  is the number of edges.
- O(n) auxiliary stack space for toposort().

*/

#include <algorithm>
#include <stdexcept>
#include <vector>

const int MAXN = 100;
std::vector<int> adj[MAXN], res;
std::vector<bool> visit(MAXN), done(MAXN);

void dfs(int u) {
  if (visit[u]) {
    throw std::runtime_error("Not a directed acyclic graph.");
  }
  if (done[u]) {
    return;
  }
  visit[u] = true;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    dfs(adj[u][j]);
  }
  visit[u] = false;
  done[u] = true;
  res.push_back(u);
}

void toposort(int nodes) {
  fill(visit.begin(), visit.end(), false);
  fill(done.begin(), done.end(), false);
  res.clear();
  for (int i = 0; i < nodes; i++) {
    if (!done[i]) {
      dfs(i);
    }
  }
  std::reverse(res.begin(), res.end());
}

/*** Example Usage and Output:

The topological order: 2 1 0 4 3 7 6 5

***/

#include <iostream>
using namespace std;
```

```
int main() {
  adj[0].push_back(3);
  adj[0].push_back(4);
  adj[1].push_back(3);
  adj[2].push_back(4);
  adj[2].push_back(7);
  adj[3].push_back(5);
  adj[3].push_back(6);
  adj[3].push_back(7);
  adj[4].push_back(6);
  toposort(8);
  cout << "The topological order:";
  for (int i = 0; i < (int)res.size(); i++) {
    cout << " " << res[i];
  }
  cout << endl;
  return 0;
}

/*

A Eulerian trail is a path in a graph which contains every edge
exactly once. An
Eulerian cycle or circuit is an Eulerian trail which begins and
ends on the same
node. A directed graph has an Eulerian cycle if and only if every
node has an
in-degree equal to its out-degree, and all of its nodes with
nonzero degree
belong to a single strongly connected component. An undirected
graph has an
Eulerian cycle if and only if every node has even degree, and all
of its nodes
with nonzero degree belong to a single connected component.

Given a graph as an adjacency list along with the starting node of
the cycle,
both functions below return a vector containing all nodes
reachable from the
starting node in an order which forms an Eulerian cycle. The first
node of the
cycle will be repeated as the last element of the vector. All
nodes of input
adjacency lists to both functions must be be between 0 and MAXN -
1, inclusive.
In addition, euler_cycle_undirected() requires that for every node
v which is
found in adj[u], node u must also be found in adj[v].

Time Complexity:
- O(max(n, m)) per call to either function, where n and m are the
```

```
numbers of
  nodes and edges respectively.

Space Complexity:
- O(n) auxiliary heap space for euler_cycle_directed(), where n is
the number of
  nodes.
- O(n^2) auxiliary heap space for euler_cycle_undirected(), where
n is the
  number of nodes. This can be reduced to O(m) auxiliary heap
space on the
  number of edges if the used[][] bit matrix is replaced with an
  std::unordered_set<std::pair<int, int>>.

*/

#include <algorithm>
#include <bitset>
#include <vector>

const int MAXN = 100;

std::vector<int> euler_cycle_directed(std::vector<int> adj[], int
u) {
  std::vector<int> stack, curr_edge(MAXN), res;
  stack.push_back(u);
  while (!stack.empty()) {
    u = stack.back();
    stack.pop_back();
    while (curr_edge[u] < (int)adj[u].size()) {
      stack.push_back(u);
      u = adj[u][curr_edge[u]++];
    }
    res.push_back(u);
  }
  std::reverse(res.begin(), res.end());
  return res;
}

std::vector<int> euler_cycle_undirected(std::vector<int> adj[],
int u) {
  std::bitset<MAXN> used[MAXN];
  std::vector<int> stack, curr_edge(MAXN), res;
  stack.push_back(u);
  while (!stack.empty()) {
    u = stack.back();
    stack.pop_back();
    while (curr_edge[u] < (int)adj[u].size()) {
      int v = adj[u][curr_edge[u]++];
      int mn = std::min(u, v), mx = std::max(u, v);
      if (!used[mn][mx]) {
```

```
        used[mn][mx] = true;
        stack.push_back(u);
        u = v;
      }
    }
    res.push_back(u);
  }
  std::reverse(res.begin(), res.end());
  return res;
}

/*** Example Usage and Output:

Eulerian cycle from 0 (directed): 0 1 3 4 1 2 0
Eulerian cycle from 2 (undirected): 2 1 3 4 1 0 2

***/

#include <iostream>
using namespace std;

int main() {
  {
    vector<int> g[5], cycle;
    g[0].push_back(1);
    g[1].push_back(2);
    g[2].push_back(0);
    g[1].push_back(3);
    g[3].push_back(4);
    g[4].push_back(1);
    cycle = euler_cycle_directed(g, 0);
    cout << "Eulerian cycle from 0 (directed):";
    for (int i = 0; i < (int)cycle.size(); i++) {
      cout << " " << cycle[i];
    }
    cout << endl;
  }
  {
    vector<int> g[5], cycle;
    g[0].push_back(1);
    g[1].push_back(0);
    g[1].push_back(2);
    g[2].push_back(1);
    g[2].push_back(0);
    g[0].push_back(2);
    g[1].push_back(3);
    g[3].push_back(1);
    g[3].push_back(4);
    g[4].push_back(3);
    g[4].push_back(1);
    g[1].push_back(4);
```

```
    cycle = euler_cycle_undirected(g, 2);
    cout << "Eulerian cycle from 2 (undirected):";
    for (int i = 0; i < (int)cycle.size(); i++) {
      cout << " " << cycle[i];
    }
    cout << endl;
  }
  return 0;
}

/*

An unweighted tree possesses a center, centroid, and diameter. The
following
functions apply to a global, pre-populated adjacency list adj[]
which satisfies
the precondition that for every node v in adj[u], node u also
exists in adj[v].
Nodes in adj[] must be numbered with integers between 0
(inclusive) and the
total number of nodes (exclusive), as passed in the function
arguments.

- find_centers() returns a vector of either one or two tree Jordan
centers. The
  Jordan center of a tree is the set of all nodes with minimum
eccentricity,
  that is, the set of all nodes where the maximum distance to all
other nodes in
  the tree is minimal.
- find_centroid() returns the node where all of its subtrees have
a size less
  than or equal to n/2, where n is the number of nodes in the
tree.
- diameter() returns the maximum distance between any two nodes in
the tree,
  using a well-known double depth-first search technique.

Time Complexity:
- O(max(n, m)) per call to find_centers(), find_centroid(), and
diameter(),
  where n is the number of nodes and m is the number of edges.

Space Complexity:
- O(n) auxiliary stack space for find_centers(), find_centroid(),
and
  diameter(), where n is the number of nodes.

*/

#include <utility>
```

```cpp
#include <vector>

const int MAXN = 100;
std::vector<int> adj[MAXN];

std::vector<int> find_centers(int nodes) {
  std::vector<int> leaves, degree(nodes);
  for (int i = 0; i < nodes; i++) {
    degree[i] = adj[i].size();
    if (degree[i] <= 1) {
      leaves.push_back(i);
    }
  }
  int removed = leaves.size();
  while (removed < nodes) {
    std::vector<int> nleaves;
    for (int i = 0; i < (int)leaves.size(); i++) {
      int u = leaves[i];
      for (int j = 0; j < (int)adj[u].size(); j++) {
        int v = adj[u][j];
        if (--degree[v] == 1) {
          nleaves.push_back(v);
        }
      }
    }
    leaves = nleaves;
    removed += leaves.size();
  }
  return leaves;
}

int find_centroid(int nodes, int u = 0, int p = -1) {
  int count = 1;
  bool good_center = true;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    int v = adj[u][j];
    if (v == p) {
      continue;
    }
    int res = find_centroid(nodes, v, u);
    if (res >= 0) {
      return res;
    }
    int size = -res;
    good_center &= (size <= nodes / 2);
    count += size;
  }
  good_center &= (nodes - count <= nodes / 2);
  return good_center ? u : -count;
}
```

```
std::pair<int, int> dfs(int u, int p, int depth) {
  std::pair<int, int> res = std::make_pair(depth, u);
  for (int j = 0; j < (int)adj[u].size(); j++) {
    if (adj[u][j] != p) {
      res = max(res, dfs(adj[u][j], u, depth + 1));
    }
  }
  return res;
}

int diameter() {
  int furthest_node = dfs(0, -1, 0).second;
  return dfs(furthest_node, -1, 0).first;
}

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
  int nodes = 6;
  adj[0].push_back(1);
  adj[1].push_back(0);
  adj[1].push_back(2);
  adj[2].push_back(1);
  adj[1].push_back(4);
  adj[4].push_back(1);
  adj[3].push_back(4);
  adj[4].push_back(3);
  adj[4].push_back(5);
  adj[5].push_back(4);
  vector<int> centers = find_centers(nodes);
  assert(centers.size() == 2 && centers[0] == 1 && centers[1] ==
4);
  assert(find_centroid(nodes) == 4);
  assert(diameter() == 3);
  return 0;
}

/*

Given a starting node in an unweighted, directed graph, visit
every connected
node and determine the minimum distance to each such node.
Optionally, output
the shortest path to a specific destination node using the
shortest-path tree
from the predecessor array pred[]. bfs() applies to a global, pre-
populated
adjacency list adj[] which consists of only nodes numbered with
```

```
integers between
0 (inclusive) and the total number of nodes (exclusive), as passed
in the
function argument.

Time Complexity:
- O(n) per call to bfs(), where n is the number of nodes.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of
nodes and m
  is the number of edges.
- O(n) auxiliary heap space for bfs().

*/

#include <queue>
#include <utility>
#include <vector>

const int MAXN = 100, INF = 0x3f3f3f3f;
std::vector<int> adj[MAXN];
int dist[MAXN], pred[MAXN];

void bfs(int nodes, int start) {
  std::vector<bool> visit(nodes, false);
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  std::queue<std::pair<int, int> > q;
  q.push(std::make_pair(start, 0));
  while (!q.empty()) {
    int u = q.front().first;
    int d = q.front().second;
    q.pop();
    visit[u] = true;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = adj[u][j];
      if (visit[v]) {
        continue;
      }
      dist[v] = d + 1;
      pred[v] = u;
      q.push(std::make_pair(v, d + 1));
    }
  }
}

/*** Example Usage and Output:
```

```
The shortest distance from 0 to 3 is 2.
Take the path: 0->1->3.

***/

#include <iostream>
using namespace std;

void print_path(int dest) {
  vector<int> path;
  for (int j = dest; pred[j] != -1; j = pred[j]) {
    path.push_back(pred[j]);
  }
  cout << "Take the path: ";
  while (!path.empty()) {
    cout << path.back() << "->";
    path.pop_back();
  }
  cout << dest << "." << endl;
}

int main() {
  int start = 0, dest = 3;
  adj[0].push_back(1);
  adj[0].push_back(3);
  adj[1].push_back(2);
  adj[1].push_back(3);
  adj[2].push_back(3);
  adj[0].push_back(3);
  bfs(4, start);
  cout << "The shortest distance from " << start << " to " << dest
<< " is "
       << dist[dest] << "." << endl;
  print_path(dest);
  return 0;
}

/*

Given a starting node in a weighted, directed graph with
nonnegative weights
only, traverse to every connected node and determine the minimum
distance to
each. Optionally, output the shortest path to a specific
destination node using
the shortest-path tree from the predecessor array pred[].
dijkstra() applies to
a global, pre-populated adjacency list adj[] which must only
consist of nodes
numbered with integers between 0 (inclusive) and the total number
of nodes
```

```
(exclusive), as passed in the function argument.

Since std::priority_queue is by default a max-heap, we simulate a
min-heap by
negating node distances before pushing them and negating them
again after
popping them. Alternatively, the container can be declared with
the following
template arguments (#include <functional> to access std::greater):
  priority_queue<pair<int, int>, vector<pair<int, int> >,
                 greater<pair<int, int> > > pq;

Dijkstra's algorithm may be modified to support negative edge
weights by
allowing nodes to be re-visited (removing the visited array check
in the inner
for-loop). This is known as the Shortest Path Faster Algorithm
(SPFA), which has
a larger running time of O(n*m) on the number of nodes and edges
respectively.
While it is as slow in the worst case as the Bellman-Ford
algorithm, the SPFA
still tends to outperform in the average case.

Time Complexity:
- O(m log n) for dijkstra(), where m is the number of edges and n
is the number
  of nodes.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of
nodes and m
  is the number of edges.
- O(n) auxiliary heap space for dijkstra().

*/

#include <limits>
#include <queue>
#include <utility>
#include <vector>

const int MAXN = 100;
std::vector<std::pair<int, int> > adj[MAXN];
int dist[MAXN], pred[MAXN];

void dijkstra(int nodes, int start) {
  std::vector<bool> visit(nodes, false);
  for (int i = 0; i < nodes; i++) {
    dist[i] = std::numeric_limits<int>::max();
    pred[i] = -1;
```

```
  }
  dist[start] = 0;
  std::priority_queue<std::pair<int, int> > pq;
  pq.push(std::make_pair(0, start));
  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();
    visit[u] = true;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = adj[u][j].first;
      if (visit[v]) {
        continue;
      }
      if (dist[v] > dist[u] + adj[u][j].second) {
        dist[v] = dist[u] + adj[u][j].second;
        pred[v] = u;
        pq.push(std::make_pair(-dist[v], v));
      }
    }
  }
}

/*** Example Usage and Output:

The shortest distance from 0 to 3 is 5.
Take the path: 0->1->2->3.

***/

#include <iostream>
using namespace std;

void print_path(int dest) {
  vector<int> path;
  for (int j = dest; pred[j] != -1; j = pred[j]) {
    path.push_back(pred[j]);
  }
  cout << "Take the path: ";
  while (!path.empty()) {
    cout << path.back() << "->";
    path.pop_back();
  }
  cout << dest << "." << endl;
}

int main() {
  int start = 0, dest = 3;
  adj[0].push_back(make_pair(1, 2));
  adj[0].push_back(make_pair(3, 8));
  adj[1].push_back(make_pair(2, 2));
  adj[1].push_back(make_pair(3, 4));
```

```
  adj[2].push_back(make_pair(3, 1));
  dijkstra(4, start);
  cout << "The shortest distance from " << start << " to " << dest
<< " is "
       << dist[dest] << "." << endl;
  print_path(dest);
  return 0;
}

/*

Given a starting node in a weighted, directed graph with possibly
negative
weights, traverse to every connected node and determine the
minimum distance to
each. Optionally, output the shortest path to a specific
destination node using
the shortest-path tree from the predecessor array pred[].
bellman_ford() applies
to a global, pre-populated edge list which must only consist of
nodes numbered
with integers between 0 (inclusive) and the total number of nodes
(exclusive),
as passed in the function argument.

This function will also detect whether the graph contains
negative-weighted
cycles, in which case there is no shortest path and an error will
be thrown.

Time Complexity:
- O(n*m) per call to bellman_ford(), where n is the number of
nodes and m is the
  number of edges.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of
nodes and m is
  the number of edges.
- O(n) auxiliary heap space for bellman_ford(), where n is the
number of nodes.

*/

#include <stdexcept>
#include <vector>

struct edge { int u, v, w; };  // Edge from u to v with weight w.

const int MAXN = 100, INF = 0x3f3f3f3f;
std::vector<edge> e;
```

```
int dist[MAXN], pred[MAXN];

void bellman_ford(int nodes, int start) {
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  dist[start] = 0;
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < (int)e.size(); j++) {
      if (dist[e[j].v] > dist[e[j].u] + e[j].w) {
        dist[e[j].v] = dist[e[j].u] + e[j].w;
        pred[e[j].v] = e[j].u;
      }
    }
  }
  // Optional: Report negative-weighted cycles.
  for (int i = 0; i < (int)e.size(); i++) {
    if (dist[e[i].v] > dist[e[i].u] + e[i].w) {
      throw std::runtime_error("Negative-weight cycle found.");
    }
  }
}

/*** Example Usage and Output:

The shortest distance from 0 to 2 is 3.
Take the path: 0->1->2.

***/

#include <iostream>
using namespace std;

void print_path(int dest) {
  vector<int> path;
  for (int j = dest; pred[j] != -1; j = pred[j]) {
    path.push_back(pred[j]);
  }
  cout << "Take the path: ";
  while (!path.empty()) {
    cout << path.back() << "->";
    path.pop_back();
  }
  cout << dest << "." << endl;
}

int main() {
  int start = 0, dest = 2;
  e.push_back((edge){0, 1, 1});
  e.push_back((edge){1, 2, 2});
```

```
  e.push_back((edge){0, 2, 5});
  bellman_ford(3, start);
  cout << "The shortest distance from " << start << " to " << dest
<< " is "
       << dist[dest] << "." << endl;
  print_path(dest);
  return 0;
}

/*

Given a weighted, directed graph with possibly negative weights,
determine the
minimum distance between all pairs of start and destination nodes
in the graph.
Optionally, output the shortest path between two nodes using the
shortest-path
tree precomputed into the parent[][] array. floyd_warshall()
applies to a global
adjacency matrix dist[][], which must be initialized using
initialize() and
subsequently populated with weights. After the function call,
dist[u][v] will
have been modified to contain the shortest path from u to v, for
all pairs of
valid nodes u and v.

This function will also detect whether the graph contains
negative-weighted
cycles, in which case there is no shortest path and an error will
be thrown.

Time Complexity:
- O(n^2) per call to initialize(), where n is the number of nodes.
- O(n^3) per call to floyd_warshall().

Space Complexity:
- O(n^2) for storage of the graph, where n is the number of nodes.
- O(n^2) auxiliary heap space for initialize() and
floyd_warshall().

*/

#include <stdexcept>

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN][MAXN], parent[MAXN][MAXN];

void initialize(int nodes) {
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < nodes; j++) {
```

```
      dist[i][j] = (i == j) ? 0 : INF;
      parent[i][j] = j;
    }
  }
}

void floyd_warshall(int nodes) {
  for (int k = 0; k < nodes; k++) {
    for (int i = 0; i < nodes; i++) {
      for (int j = 0; j < nodes; j++) {
        if (dist[i][j] > dist[i][k] + dist[k][j]) {
          dist[i][j] = dist[i][k] + dist[k][j];
          parent[i][j] = parent[i][k];
        }
      }
    }
  }
  // Optional: Report negative-weighted cycles.
  for (int i = 0; i < nodes; i++) {
    if (dist[i][i] < 0) {
      throw std::runtime_error("Negative-weight cycle found.");
    }
  }
}

/*** Example Usage and Output:

The shortest distance from 0 to 2 is 3.
Take the path: 0->1->2.

***/

#include <iostream>
using namespace std;

void print_path(int u, int v) {
  cout << "Take the path " << u;
  while (u != v) {
    u = parent[u][v];
    cout << "->" << u;
  }
  cout << "." << endl;
}

int main() {
  initialize(3);
  int start = 0, dest = 2;
  dist[0][1] = 1;
  dist[1][2] = 2;
  dist[0][2] = 5;
  floyd_warshall(3);
```

```
  cout << "The shortest distance from " << start << " to " << dest
<< " is "
       << dist[start][dest] << "." << endl;
  print_path(start, dest);
  return 0;
}

/*

Given a weighted, directed graph with possibly negative weights,
determine the
minimum distance between all pairs of start and destination nodes
in the graph.
Optionally, output the shortest path between two nodes using the
shortest-path
tree precomputed into the parent[][] array. floyd_warshall()
applies to a global
adjacency matrix dist[][], which must be initialized using
initialize() and
subsequently populated with weights. After the function call,
dist[u][v] will
have been modified to contain the shortest path from u to v, for
all pairs of
valid nodes u and v.

This function will also detect whether the graph contains
negative-weighted
cycles, in which case there is no shortest path and an error will
be thrown.

Time Complexity:
- O(n^2) per call to initialize(), where n is the number of nodes.
- O(n^3) per call to floyd_warshall().

Space Complexity:
- O(n^2) for storage of the graph, where n is the number of nodes.
- O(n^2) auxiliary heap space for initialize() and
floyd_warshall().

*/

#include <stdexcept>

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN][MAXN], parent[MAXN][MAXN];

void initialize(int nodes) {
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < nodes; j++) {
      dist[i][j] = (i == j) ? 0 : INF;
      parent[i][j] = j;
```

```
    }
  }
}

void floyd_warshall(int nodes) {
  for (int k = 0; k < nodes; k++) {
    for (int i = 0; i < nodes; i++) {
      for (int j = 0; j < nodes; j++) {
        if (dist[i][j] > dist[i][k] + dist[k][j]) {
          dist[i][j] = dist[i][k] + dist[k][j];
          parent[i][j] = parent[i][k];
        }
      }
    }
  }
  // Optional: Report negative-weighted cycles.
  for (int i = 0; i < nodes; i++) {
    if (dist[i][i] < 0) {
      throw std::runtime_error("Negative-weight cycle found.");
    }
  }
}

/*** Example Usage and Output:

The shortest distance from 0 to 2 is 3.
Take the path: 0->1->2.

***/

#include <iostream>
using namespace std;

void print_path(int u, int v) {
  cout << "Take the path " << u;
  while (u != v) {
    u = parent[u][v];
    cout << "->" << u;
  }
  cout << "." << endl;
}

int main() {
  initialize(3);
  int start = 0, dest = 2;
  dist[0][1] = 1;
  dist[1][2] = 2;
  dist[0][2] = 5;
  floyd_warshall(3);
  cout << "The shortest distance from " << start << " to " << dest
<< " is "
```

```
       << dist[start][dest] << "." << endl;
  print_path(start, dest);
  return 0;
}

/*

Given a connected, undirected, weighted graph with possibly
negative weights,
its minimum spanning tree is a subgraph which is a tree that
connects all nodes
with a subset of its edges such that their total weight is
minimized. kruskal()
applies to a global, pre-populated adjacency list adj[] which must
only consist
of nodes numbered with integers between 0 (inclusive) and the
total number of
nodes (exclusive), as passed in the function argument. If the
input graph is not
connected, then this implementation will find the minimum spanning
forest.

Time Complexity:
- O(m log n) per call to kruskal(), where m is the number of edges
and n is the
  number of nodes.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n the number of
nodes and m is
  the number of edges
- O(n) auxiliary stack space for kruskal().

*/

#include <algorithm>
#include <utility>
#include <vector>

const int MAXN = 100;
std::vector<std::pair<int, std::pair<int, int> > > edges;
int root[MAXN];
std::vector<std::pair<int, int> > mst;

int find_root(int x) {
  if (root[x] != x) {
    root[x] = find_root(root[x]);
  }
  return root[x];
}
```

```
int kruskal(int nodes) {
  mst.clear();
  std::sort(edges.begin(), edges.end());
  int total_dist = 0;
  for (int i = 0; i < nodes; i++) {
    root[i] = i;
  }
  for (int i = 0; i < (int)edges.size(); i++) {
    int u = find_root(edges[i].second.first);
    int v = find_root(edges[i].second.second);
    if (u != v) {
      root[u] = root[v];
      mst.push_back(edges[i].second);
      total_dist += edges[i].first;
    }
  }
  return total_dist;
}

/*** Example Usage and Output:

Total distance: 13
3 <-> 4
4 <-> 5
2 <-> 0
5 <-> 6
0 <-> 1

***/

#include <iostream>
using namespace std;

void add_edge(int u, int v, int w) {
  edges.push_back(make_pair(w, make_pair(u, v)));
}

int main() {
  add_edge(0, 1, 4);
  add_edge(1, 2, 6);
  add_edge(2, 0, 3);
  add_edge(3, 4, 1);
  add_edge(4, 5, 2);
  add_edge(5, 6, 3);
  add_edge(6, 4, 4);
  cout << "Total distance: " << kruskal(7) << endl;
  for (int i = 0; i < (int)mst.size(); i++) {
    cout << mst[i].first << " <-> " << mst[i].second << endl;
  }
  return 0;
}
```

```
/*

Given a flow network with integer capacities, find the maximum
flow from a given
source node to a given sink node. The flow of a given edge u -> v
is defined as
the minimum of its capacity and the sum of the flows of all
incoming edges of u.
ford_fulkerson() applies to global variables nodes, source, sink,
and cap[][]
which is an adjacency matrix that will be modified by the function
call.

The Ford-Fulkerson algorithm is only optimal on graphs with
integer capacities,
as there exists certain real-valued flow inputs for which the
algorithm never
terminates. The Edmonds-Karp algorithm is an improvement using
breadth-first
search, addressing this problem.

Time Complexity:
- O(n^2*f) per call to ford_fulkerson(), where n is the number of
nodes and f
  is the maximum flow.

Space Complexity:
- O(n^2) for storage of the flow network, where n is the number of
nodes.
- O(n) auxiliary stack space for ford_fulkerson().

*/

#include <algorithm>
#include <vector>

const int MAXN = 100, INF = 0x3f3f3f3f;
int nodes, source, sink, cap[MAXN][MAXN];
std::vector<bool> visit(MAXN);

int dfs(int u, int f) {
  if (u == sink) {
    return f;
  }
  visit[u] = true;
  for (int v = 0; v < nodes; v++) {
    if (!visit[v] && cap[u][v] > 0) {
      int flow = dfs(v, std::min(f, cap[u][v]));
      if (flow > 0) {
        cap[u][v] -= flow;
```

```
        cap[v][u] += flow;
        return flow;
      }
    }
  }
  return 0;
}

int ford_fulkerson() {
  int max_flow = 0;
  for (;;) {
    std::fill(visit.begin(), visit.end(), false);
    int flow = dfs(source, INF);
    if (flow == 0) {
      break;
    }
    max_flow += flow;
  }
  return max_flow;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  nodes = 6;
  source = 0;
  sink = 5;
  cap[0][1] = 3;
  cap[0][2] = 3;
  cap[1][2] = 2;
  cap[1][3] = 3;
  cap[2][4] = 2;
  cap[3][4] = 1;
  cap[3][5] = 2;
  cap[4][5] = 3;
  assert(ford_fulkerson() == 5);
  return 0;
}

/*

Given a flow network with integer capacities, find the maximum
flow from a given
source node to a given sink node. The flow of a given edge u -> v
is defined as
the minimum of its capacity and the sum of the flows of all
incoming edges of u.
ford_fulkerson() applies to global variables nodes, source, sink,
and cap[][]
```

```
which is an adjacency matrix that will be modified by the function
call.

The Ford-Fulkerson algorithm is only optimal on graphs with
integer capacities,
as there exists certain real-valued flow inputs for which the
algorithm never
terminates. The Edmonds-Karp algorithm is an improvement using
breadth-first
search, addressing this problem.

Time Complexity:
- O(n^2*f) per call to ford_fulkerson(), where n is the number of
nodes and f
  is the maximum flow.

Space Complexity:
- O(n^2) for storage of the flow network, where n is the number of
nodes.
- O(n) auxiliary stack space for ford_fulkerson().

*/

#include <algorithm>
#include <vector>

const int MAXN = 100, INF = 0x3f3f3f3f;
int nodes, source, sink, cap[MAXN][MAXN];
std::vector<bool> visit(MAXN);

int dfs(int u, int f) {
  if (u == sink) {
    return f;
  }
  visit[u] = true;
  for (int v = 0; v < nodes; v++) {
    if (!visit[v] && cap[u][v] > 0) {
      int flow = dfs(v, std::min(f, cap[u][v]));
      if (flow > 0) {
        cap[u][v] -= flow;
        cap[v][u] += flow;
        return flow;
      }
    }
  }
  return 0;
}

int ford_fulkerson() {
  int max_flow = 0;
  for (;;) {
```

```
    std::fill(visit.begin(), visit.end(), false);
    int flow = dfs(source, INF);
    if (flow == 0) {
      break;
    }
    max_flow += flow;
  }
  return max_flow;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  nodes = 6;
  source = 0;
  sink = 5;
  cap[0][1] = 3;
  cap[0][2] = 3;
  cap[1][2] = 2;
  cap[1][3] = 3;
  cap[2][4] = 2;
  cap[3][4] = 1;
  cap[3][5] = 2;
  cap[4][5] = 3;
  assert(ford_fulkerson() == 5);
  return 0;
}

/*

Maintain a map, that is, a collection of key-value pairs such that
each possible
key appears at most once in the collection. This implementations
requires an
ordering on the set of possible keys defined by the < operator on
the key type.
A binary search tree implements this map by inserting and deleting
keys into a
binary tree such that every node's left child has a lesser key and
every node's
right child has a greater key.

- binary_search_tree() constructs an empty map.
- size() returns the size of the map.
- empty() returns the map is empty.
- insert(k, v) adds an entry with key k and value v to the map,
returning true
  if an new entry was added or false if the key already exists (in
which case
```

```
  the map is unchanged and the old value associated with the key
is preserved).
- erase(k) removes the entry with key k from the map, returning
true if the
  removal was successful or false if the key to be removed was not
found.
- find(k) returns a pointer to a const value associated with key
k, or NULL if
  the key was not found.
- walk(f) calls the function f(k, v) on each entry of the map, in
ascending
  order of keys.

Time Complexity:
- O(1) per call to the constructor, size(), and empty().
- O(n) per call to insert(), erase(), find(), and walk(), where n
is the number
  of nodes currently in the map.

Space Complexity:
- O(n) for storage of the map elements.
- O(n) auxiliary stack space for insert(), erase(), and walk().
- O(1) auxiliary for all other operations.

*/

#include <cstdlib>

template<class K, class V> class binary_search_tree {
  struct node_t {
    K key;
    V value;
    node_t *left, *right;

    node_t(const K &k, const V &v)
        : key(k), value(v), left(NULL), right(NULL) {}
  } *root;

  int num_nodes;

  static bool insert(node_t *&n, const K &k, const V &v) {
    if (n == NULL) {
      n = new node_t(k, v);
      return true;
    }
    if (k < n->key) {
      return insert(n->left, k, v);
    } else if (n->key < k) {
      return insert(n->right, k, v);
    }
    return false;
```

```
  }

  static bool erase(node_t *&n, const K &k) {
    if (n == NULL) {
      return false;
    }
    if (k < n->key) {
      return erase(n->left, k);
    } else if (n->key < k) {
      return erase(n->right, k);
    }
    if (n->left != NULL && n->right != NULL) {
      node_t *tmp = n->right, *parent = NULL;
      while (tmp->left != NULL) {
        parent = tmp;
        tmp = tmp->left;
      }
      n->key = tmp->key;
      n->value = tmp->value;
      if (parent != NULL) {
        return erase(parent->left, parent->left->key);
      }
      return erase(n->right, n->right->key);
    }
    node_t *tmp = (n->left != NULL) ? n->left : n->right;
    delete n;
    n = tmp;
    return true;
  }

  template<class KVFunction>
  static void walk(node_t *n, KVFunction f) {
    if (n != NULL) {
      walk(n->left, f);
      f(n->key, n->value);
      walk(n->right, f);
    }
  }

  static void clean_up(node_t *n) {
    if (n != NULL) {
      clean_up(n->left);
      clean_up(n->right);
      delete n;
    }
  }

 public:
  binary_search_tree() : root(NULL), num_nodes(0) {}

  ~binary_search_tree() {
```

```
    clean_up(root);
  }

  int size() const {
    return num_nodes;
  }

  bool empty() const {
    return root == NULL;
  }

  bool insert(const K &k, const V &v) {
    if (insert(root, k, v)) {
      num_nodes++;
      return true;
    }
    return false;
  }

  bool erase(const K &k) {
    if (erase(root, k)) {
      num_nodes--;
      return true;
    }
    return false;
  }

  const V* find(const K &k) const {
    node_t *n = root;
    while (n != NULL) {
      if (k < n->key) {
        n = n->left;
      } else if (n->key < k) {
        n = n->right;
      } else {
        return &(n->value);
      }
    }
    return NULL;
  }

  template<class KVFunction>
  void walk(KVFunction f) const {
    walk(root, f);
  }
};

/*** Example Usage and Output:

abcde
bcde
```

```
***/

#include <cassert>
#include <iostream>
using namespace std;

void printch(int k, char v) {
  cout << v;
}

int main() {
  binary_search_tree<int, char> t;
  t.insert(2, 'b');
  t.insert(1, 'a');
  t.insert(3, 'c');
  t.insert(5, 'e');
  assert(t.insert(4, 'd'));
  assert(*t.find(4) == 'd');
  assert(!t.insert(4, 'd'));
  t.walk(printch);
  cout << endl;
  assert(t.erase(1));
  assert(!t.erase(1));
  assert(t.find(1) == NULL);
  t.walk(printch);
  cout << endl;
  return 0;
}

/*

Maintain a map, that is, a collection of key-value pairs such that
each possible
key appears at most once in the collection. This implementation
requires an
ordering on the set of possible keys defined by the < operator on
the key type.
An AVL tree is a binary search tree balanced by height,
guaranteeing O(log n)
worst-case running time in insertions and deletions by making sure
that the
heights of the left and right subtrees at every node differ by at
most 1.

- avl_tree() constructs an empty map.
- size() returns the size of the map.
- empty() returns whether the map is empty.
- insert(k, v) adds an entry with key k and value v to the map,
returning true
  if an new entry was added or false if the key already exists (in
```

```
which case
  the map is unchanged and the old value associated with the key
is preserved).
- erase(k) removes the entry with key k from the map, returning
true if the
  removal was successful or false if the key to be removed was not
found.
- find(k) returns a pointer to a const value associated with key
k, or NULL if
  the key was not found.
- walk(f) calls the function f(k, v) on each entry of the map, in
ascending
  order of keys.

Time Complexity:
- O(1) per call to the constructor, size(), and empty().
- O(log n) per call to insert(), erase(), and find(), where n is
the number of
  entries currently in the map.
- O(n) per call to walk().

Space Complexity:
- O(n) for storage of the map elements.
- O(log n) auxiliary stack space for insert(), erase(), and
walk().
- O(1) auxiliary for all other operations.

*/

#include <algorithm>
#include <cstdlib>

template<class K, class V> class avl_tree {
  struct node_t {
    K key;
    V value;
    int height;
    node_t *left, *right;

    node_t(const K &k, const V &v)
        : key(k), value(v), height(1), left(NULL), right(NULL) {}
  } *root;

  int num_nodes;

  static int height(node_t *n) {
    return (n != NULL) ? n->height : 0;
  }

  static void update_height(node_t *n) {
    if (n != NULL) {
```

```
    n->height = 1 + std::max(height(n->left), height(n->right));
  }
}

static void rotate_left(node_t *&n) {
  node_t *tmp = n;
  n = n->right;
  tmp->right = n->left;
  n->left = tmp;
  update_height(tmp);
  update_height(n);
}

static void rotate_right(node_t *&n) {
  node_t *tmp = n;
  n = n->left;
  tmp->left = n->right;
  n->right = tmp;
  update_height(tmp);
  update_height(n);
}

static int balance_factor(node_t *n) {
  return (n != NULL) ? (height(n->left) - height(n->right)) : 0;
}

static void rebalance(node_t *&n) {
  if (n == NULL) {
    return;
  }
  update_height(n);
  int bf = balance_factor(n);
  if (bf > 1 && balance_factor(n->left) >= 0) {
    rotate_right(n);
  } else if (bf > 1 && balance_factor(n->left) < 0) {
    rotate_left(n->left);
    rotate_right(n);
  } else if (bf < -1 && balance_factor(n->right) <= 0) {
    rotate_left(n);
  } else if (bf < -1 && balance_factor(n->right) > 0) {
    rotate_right(n->right);
    rotate_left(n);
  }
}

static bool insert(node_t *&n, const K &k, const V &v) {
  if (n == NULL) {
    n = new node_t(k, v);
    return true;
  }
  if ((k < n->key && insert(n->left, k, v)) ||
```

```
      (n->key < k && insert(n->right, k, v))) {
    rebalance(n);
    return true;
  }
  return false;
}

static bool erase(node_t *&n, const K &k) {
  if (n == NULL) {
    return false;
  }
  if (!(k < n->key || n->key < k)) {
    if (n->left != NULL && n->right != NULL) {
      node_t *tmp = n->right, *parent = NULL;
      while (tmp->left != NULL) {
        parent = tmp;
        tmp = tmp->left;
      }
      n->key = tmp->key;
      n->value = tmp->value;
      if (parent != NULL) {
        if (!erase(parent->left, parent->left->key)) {
          return false;
        }
      } else if (!erase(n->right, n->right->key)) {
        return false;
      }
    } else {
      node_t *tmp = (n->left != NULL) ? n->left : n->right;
      delete n;
      n = tmp;
    }
    rebalance(n);
    return true;
  }
  if ((k < n->key && erase(n->left, k)) ||
      (n->key < k && erase(n->right, k))) {
    rebalance(n);
    return true;
  }
  return false;
}

template<class KVFunction>
static void walk(node_t *n, KVFunction f) {
  if (n != NULL) {
    walk(n->left, f);
    f(n->key, n->value);
    walk(n->right, f);
  }
}
```

```
  static void clean_up(node_t *n) {
    if (n != NULL) {
      clean_up(n->left);
      clean_up(n->right);
      delete n;
    }
  }

public:
  avl_tree() : root(NULL), num_nodes(0) {}

  ~avl_tree() {
    clean_up(root);
  }

  int size() const {
    return num_nodes;
  }

  bool empty() const {
    return root == NULL;
  }

  bool insert(const K &k, const V &v) {
    if (insert(root, k, v)) {
      num_nodes++;
      return true;
    }
    return false;
  }

  bool erase(const K &k) {
    if (erase(root, k)) {
      num_nodes--;
      return true;
    }
    return false;
  }

  const V* find(const K &k) const {
    node_t *n = root;
    while (n != NULL) {
      if (k < n->key) {
        n = n->left;
      } else if (n->key < k) {
        n = n->right;
      } else {
        return &(n->value);
      }
    }
```

```
    return NULL;
  }

  template<class KVFunction>
  void walk(KVFunction f) const {
    walk(root, f);
  }
};

/*** Example Usage and Output:

abcde
bcde

***/

#include <cassert>
#include <iostream>
using namespace std;

void printch(int k, char v) {
  cout << v;
}

int main() {
  avl_tree<int, char> t;
  t.insert(2, 'b');
  t.insert(1, 'a');
  t.insert(3, 'c');
  t.insert(5, 'e');
  assert(t.insert(4, 'd'));
  assert(*t.find(4) == 'd');
  assert(!t.insert(4, 'd'));
  t.walk(printch);
  cout << endl;
  assert(t.erase(1));
  assert(!t.erase(1));
  assert(t.find(1) == NULL);
  t.walk(printch);
  cout << endl;
  return 0;
}

/*

Maintain a map, that is, a collection of key-value pairs such that
each possible
key appears at most once in the collection. This implementation
requires an
ordering on the set of possible keys defined by the < operator on
the key type.
```

```
A red black tree is a binary search tree balanced by coloring its
nodes red or
black, then constraining node colors on any simple path from the
root to a leaf.

- red_black_tree() constructs an empty map.
- size() returns the size of the map.
- empty() returns whether the map is empty.
- insert(k, v) adds an entry with key k and value v to the map,
returning true
  if an new entry was added or false if the key already exists (in
which case
  the map is unchanged and the old value associated with the key
is preserved).
- erase(k) removes the entry with key k from the map, returning
true if the
  removal was successful or false if the key to be removed was not
found.
- find(k) returns a pointer to a const value associated with key
k, or NULL if
  the key was not found.
- walk(f) calls the function f(k, v) on each entry of the map, in
ascending
  order of keys.

Time Complexity:
- O(1) per call to the constructor, size(), and empty().
- O(log n) per call to insert(), erase(), and find(), where n is
the number of
  entries currently in the map.
- O(n) per call to walk().

Space Complexity:
- O(n) for storage of the map elements.
- O(log n) auxiliary stack space for walk().
- O(1) auxiliary for all other operations.

*/

#include <algorithm>
#include <cstdlib>

template<class K, class V> class red_black_tree {
  enum color_t { RED, BLACK };
  struct node_t {
    K key;
    V value;
    color_t color;
    node_t *left, *right, *parent;

    node_t(const K &k, const V &v, color_t c)
```

```
        : key(k), value(v), color(c), left(NULL), right(NULL),
parent(NULL) {}
  } *root, *LEAF_NIL;

  int num_nodes;

  void rotate_left(node_t *n) {
    node_t *tmp = n->right;
    if ((n->right = tmp->left) != LEAF_NIL) {
      n->right->parent = n;
    }
    if ((tmp->parent = n->parent) == LEAF_NIL) {
      root = tmp;
    } else if (n->parent->left == n) {
      n->parent->left = tmp;
    } else {
      n->parent->right = tmp;
    }
    tmp->left = n;
    n->parent = tmp;
  }

  void rotate_right(node_t *n) {
    node_t *tmp = n->left;
    if ((n->left = tmp->right) != LEAF_NIL) {
      n->left->parent = n;
    }
    if ((tmp->parent = n->parent) == LEAF_NIL) {
      root = tmp;
    } else if (n->parent->right == n) {
      n->parent->right = tmp;
    } else {
      n->parent->left = tmp;
    }
    tmp->right = n;
    n->parent = tmp;
  }

  void insert_fix(node_t *n) {
    while (n->parent->color == RED) {
      node_t *parent = n->parent;
      node_t *grandparent = n->parent->parent;
      if (parent == grandparent->left) {
        node_t *uncle = grandparent->right;
        if (uncle->color == RED) {
          grandparent->color = RED;
          parent->color = BLACK;
          uncle->color = BLACK;
          n = grandparent;
        } else {
          if (n == parent->right) {
```

```
          rotate_left(parent);
          n = parent;
          parent = n->parent;
        }
        rotate_right(grandparent);
        std::swap(parent->color, grandparent->color);
        n = parent;
      }
    } else if (parent == grandparent->right) {
      node_t *uncle = grandparent->left;
      if (uncle->color == RED) {
        grandparent->color = RED;
        parent->color = BLACK;
        uncle->color = BLACK;
        n = grandparent;
      } else {
        if (n == parent->left) {
          rotate_right(parent);
          n = parent;
          parent = n->parent;
        }
        rotate_left(grandparent);
        std::swap(parent->color, grandparent->color);
        n = parent;
      }
    }
  }
  root->color = BLACK;
}

void replace(node_t *n, node_t *replacement) {
  if (n->parent == LEAF_NIL) {
    root = replacement;
  } else if (n == n->parent->left) {
    n->parent->left = replacement;
  } else {
    n->parent->right = replacement;
  }
  replacement->parent = n->parent;
}

void erase_fix(node_t *n) {
  while (n != root && n->color == BLACK) {
    node_t *parent = n->parent;
    if (n == parent->left) {
      node_t *sibling = parent->right;
      if (sibling->color == RED) {
        sibling->color = BLACK;
        parent->color = RED;
        rotate_left(parent);
        sibling = parent->right;
      }
```

```
        if (sibling->left->color == BLACK && sibling->right->color
== BLACK) {
          sibling->color = RED;
          n = parent;
        } else {
          if (sibling->right->color == BLACK) {
            sibling->left->color = BLACK;
            sibling->color = RED;
            rotate_right(sibling);
            sibling = parent->right;
          }
          sibling->color = parent->color;
          parent->color = BLACK;
          sibling->right->color = BLACK;
          rotate_left(parent);
          n = root;
        }
      } else {
        node_t *sibling = parent->left;
        if (sibling->color == RED) {
          sibling->color = BLACK;
          parent->color = RED;
          rotate_right(parent);
          sibling = parent->left;
        }
        if (sibling->left->color == BLACK && sibling->right->color
== BLACK) {
          sibling->color = RED;
          n = parent;
        } else {
          if (sibling->left->color == BLACK) {
            sibling->right->color = BLACK;
            sibling->color = RED;
            rotate_left(sibling);
            sibling = parent->left;
          }
          sibling->color = parent->color;
          parent->color = BLACK;
          sibling->left->color = BLACK;
          rotate_right(parent);
          n = root;
        }
      }
    }
    n->color = BLACK;
  }

  template<class KVFunction>
  void walk(node_t *n, KVFunction f) const {
    if (n != LEAF_NIL) {
      walk(n->left, f);
      f(n->key, n->value);
```

```
      walk(n->right, f);
    }
  }

  void clean_up(node_t *n) {
    if (n != LEAF_NIL) {
      clean_up(n->left);
      clean_up(n->right);
      delete n;
    }
  }

public:
  red_black_tree() : num_nodes(0) {
    root = LEAF_NIL = new node_t(K(), V(), BLACK);
  }

  ~red_black_tree() {
    clean_up(root);
    delete LEAF_NIL;
  }

  int size() const {
    return num_nodes;
  }

  bool empty() const {
    return num_nodes == 0;
  }

  bool insert(const K &k, const V &v) {
    node_t *curr = root, *prev = LEAF_NIL;
    while (curr != LEAF_NIL) {
      prev = curr;
      if (k < curr->key) {
        curr = curr->left;
      } else if (curr->key < k) {
        curr = curr->right;
      } else {
        return false;
      }
    }
    node_t *n = new node_t(k, v, RED);
    n->parent = prev;
    if (prev == LEAF_NIL) {
      root = n;
    } else if (k < prev->key) {
      prev->left = n;
    } else {
      prev->right = n;
    }
```

```
    n->left = n->right = LEAF_NIL;
    insert_fix(n);
    num_nodes++;
    return true;
  }

  bool erase(const K &k) {
    node_t *n = root;
    while (n != LEAF_NIL) {
      if (k < n->key) {
        n = n->left;
      } else if (n->key < k) {
        n = n->right;
      } else {
        break;
      }
    }
    if (n == LEAF_NIL) {
      return false;
    }
    color_t color = n->color;
    node_t *replacement;
    if (n->left == LEAF_NIL) {
      replacement = n->right;
      replace(n, n->right);
    } else if (n->right == LEAF_NIL) {
      replacement = n->left;
      replace(n, n->left);
    } else {
      node_t *tmp = n->right;
      while (tmp->left != LEAF_NIL) {
        tmp = tmp->left;
      }
      color = tmp->color;
      replacement = tmp->right;
      if (tmp->parent == n) {
        replacement->parent = tmp;
      } else {
        replace(tmp, tmp->right);
        tmp->right = n->right;
        tmp->right->parent = tmp;
      }
      replace(n, tmp);
      tmp->left = n->left;
      tmp->left->parent = tmp;
      tmp->color = n->color;
    }
    delete n;
    if (color == BLACK) {
      erase_fix(replacement);
    }
    return true;
```

```
  }

  const V* find(const K &k) const {
    node_t *n = root;
    while (n != LEAF_NIL) {
      if (k < n->key) {
        n = n->left;
      } else if (n->key < k) {
        n = n->right;
      } else {
        return &(n->value);
      }
    }
    return NULL;
  }

  template<class KVFunction>
  void walk(KVFunction f) const {
    walk(root, f);
  }
};

/*** Example Usage and Output:

abcde
bcde

***/

#include <cassert>
#include <iostream>
using namespace std;

void printch(int k, char v) {
  cout << v;
}

int main() {
  red_black_tree<int, char> t;
  t.insert(2, 'b');
  t.insert(1, 'a');
  t.insert(3, 'c');
  t.insert(5, 'e');
  assert(t.insert(4, 'd'));
  assert(*t.find(4) == 'd');
  assert(!t.insert(4, 'd'));
  t.walk(printch);
  cout << endl;
  assert(t.erase(1));
  assert(!t.erase(1));
  assert(t.find(1) == NULL);
```

```
  t.walk(printch);
  cout << endl;
  return 0;
}

/*

Maintain a map, that is, a collection of key-value pairs such that
each possible
key appears at most once in the collection. This implementation
requires the ==
operator to be defined on the key type. A hash map implements a
map by hashing
keys into buckets using a hash function. This implementation
resolves collisions
by chaining entries hashed to the same bucket into a linked list.

- hash_map() constructs an empty map.
- size() returns the size of the map.
- empty() returns whether the map is empty.
- insert(k, v) adds an entry with key k and value v to the map,
returning true
  if an new entry was added or false if the key already exists (in
which case
  the map is unchanged and the old value associated with the key
is preserved).
- erase(k) removes the entry with key k from the map, returning
true if the
  removal was successful or false if the key to be removed was not
found.
- find(k) returns a pointer to a const value associated with key
k, or NULL if
  the key was not found.
- operator[k] returns a reference to key k's associated value
(which may be
  modified), or if necessary, inserts and returns a new entry with
the default
  constructed value if key k was not originally found.
- walk(f) calls the function f(k, v) on each entry of the map, in
no guaranteed
  order.

Time Complexity:
- O(1) per call to the constructor, size(), and empty().
- O(1) amortized per call to insert(), erase(), find(), and
operator[].
- O(n) per call to walk(), where n is the number of entries in the
map.

Space Complexity:
- O(n) for storage of the map elements.
```

```
- O(n) auxiliary heap space for insert().
- O(1) auxiliary for all other operations.

*/

#include <cstdlib>
#include <list>

template<class K, class V, class Hash> class hash_map {
  struct entry_t {
    K key;
    V value;

    entry_t(const K &k, const V &v) : key(k), value(v) {}
  };

  std::list<entry_t> *table;
  int table_size, num_entries;

  void double_capacity_and_rehash() {
    std::list<entry_t> *old = table;
    int old_size = table_size;
    table_size = 2*table_size;
    table = new std::list<entry_t>[table_size];
    num_entries = 0;
    typename std::list<entry_t>::iterator it;
    for (int i = 0; i < old_size; i++) {
      for (it = old[i].begin(); it != old[i].end(); ++it) {
        insert(it->key, it->value);
      }
    }
    delete[] old;
  }

 public:
  hash_map(int size = 128) : table_size(size), num_entries(0) {
    table = new std::list<entry_t>[table_size];
  }

  ~hash_map() {
    delete[] table;
  }

  int size() const {
    return num_entries;
  }

  bool empty() const {
    return num_entries == 0;
  }
```

```
bool insert(const K &k, const V &v) {
  if (find(k) != NULL) {
    return false;
  }
  if (num_entries >= table_size) {
    double_capacity_and_rehash();
  }
  unsigned int i = Hash()(k) % table_size;
  table[i].push_back(entry_t(k, v));
  num_entries++;
  return true;
}

bool erase(const K &k) {
  unsigned int i = Hash()(k) % table_size;
  typename std::list<entry_t>::iterator it = table[i].begin();
  while (it != table[i].end() && !(it->key == k)) {
    ++it;
  }
  if (it == table[i].end()) {
    return false;
  }
  table[i].erase(it);
  num_entries--;
  return true;
}

V* find(const K &k) const {
  unsigned int i = Hash()(k) % table_size;
  typename std::list<entry_t>::iterator it = table[i].begin();
  while (it != table[i].end() && !(it->key == k)) {
    ++it;
  }
  if (it == table[i].end()) {
    return NULL;
  }
  return &(it->value);
}

V& operator[](const K &k) {
  V *ret = find(k);
  if (ret != NULL) {
    return *ret;
  }
  insert(k, V());
  return *find(k);
}

template<class KVFunction>
void walk(KVFunction f) const {
  for (int i = 0; i < table_size; i++) {
```

```
      typename std::list<entry_t>::iterator it;
      for (it = table[i].begin(); it != table[i].end(); ++it) {
        f(it->key, it->value);
      }
    }
  }
};

/*** Example Usage and Output:

cab

***/

#include <cassert>
#include <iostream>
using namespace std;

struct class_hash {
  unsigned int operator()(int k) {
    return class_hash()((unsigned int)k);
  }

  unsigned int operator()(long long k) {
    return class_hash()((unsigned long long)k);
  }

  // Knuth's one-to-one multiplicative method.
  unsigned int operator()(unsigned int k) {
    return k * 2654435761u;  // Or just return k.
  }

  // Jenkins's 64-bit hash.
  unsigned int operator()(unsigned long long k) {
    k += ~(k << 32);
    k ^=  (k >> 22);
    k += ~(k << 13);
    k ^=  (k >> 8);
    k +=  (k << 3);
    k ^=  (k >> 15);
    k += ~(k << 27);
    k ^=  (k >> 31);
    return k;
  }

  // Jenkins's one-at-a-time hash.
  unsigned int operator()(const std::string &k) {
    unsigned int hash = 0;
    for (unsigned int i = 0; i < k.size(); i++) {
      hash += ((hash + k[i]) << 10);
      hash ^= (hash >> 6);
```

```
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    return hash + (hash << 15);
  }
};

void printch(const string &k, char v) {
  cout << v;
}

int main() {
  hash_map<string, char, class_hash> m;
  m["foo"] = 'a';
  m.insert("bar", 'b');
  assert(m["foo"] == 'a');
  assert(m["bar"] == 'b');
  assert(m["baz"] == '\0');
  m["baz"] = 'c';
  m.walk(printch);
  cout << endl;
  assert(m.erase("foo"));
  assert(m.size() == 2);
  assert(m["foo"] == '\0');
  assert(m.size() == 3);
  return 0;
}

/*

Common mathematic constants and functions, many of which are
substitutes for
features which are not available in standard C++, or may not be
available on
compilers that do not support C++11 or later.

Time Complexity:
- O(1) for all operations.

Space Complexity:
- O(1) auxiliary for all operations.

*/

#include <algorithm>
#include <cfloat>
#include <climits>
#include <cmath>
#include <cstdlib>
#include <limits>
#include <string>
```

```
#include <vector>

#ifndef M_PI
  const double M_PI = acos(-1.0);
#endif
#ifndef M_E
  const double M_E = exp(1.0);
#endif
const double M_PHI = (1.0 + sqrt(5.0))/2.0;
const double M_INF = std::numeric_limits<double>::infinity();
const double M_NAN = std::numeric_limits<double>::quiet_NaN();

#ifndef isnan
  #define isnan(x) ((x) != (x))
#endif

/*

Epsilon Comparisons

EQ(), NE(), LT(), GT(), LE(), and GE() relationally compares two
values x and y
accounting for absolute error. For any x, the range of values
considered equal
barring absolute error is [x - EPS, x + EPS]. Values outside of
this range are
considered not equal (strictly less or strictly greater).

rEQ() returns whether x and y are equal barring relative error.
For any x, the
range of values considered equal is [x*(1 - EPS), x*(1 + EPS)].

*/

const double EPS = 1e-9;

#define EQ(x, y) (fabs((x) - (y)) <= EPS)
#define NE(x, y) (fabs((x) - (y)) > EPS)
#define LT(x, y) ((x) < (y) - EPS)
#define GT(x, y) ((x) > (y) + EPS)
#define LE(x, y) ((x) <= (y) + EPS)
#define GE(x, y) ((x) >= (y) - EPS)
#define rEQ(x, y) (fabs((x) - (y)) <= EPS*fabs(x))

/*

Sign Functions

- sgn(x) returns -1 (if x < 0), 0 (if x == 0), or 1 (if x > 0).
Unlike signbit()
  or copysign(), this does not handle the sign of NaN.
```

```
- signbit_(x) is analogous to std::signbit() in C++11 and later,
returning
  whether the sign bit of the floating point number is set to
true. If so, then
  x is considered "negative." Note that this works as expected on
+0.0, -0.0,
  Inf, -Inf, NaN, as well as -NaN. The first version requires that
sizeof(int)
  equals sizeof(float) while the second version requires that
sizeof(long long)
  equals sizeof(double).
- copysign_(x, y) is analogous to std::copysign() in C++11 and
later, returning
  a number with the magnitude of x but the sign of y.

*/

template<class T>
int sgn(const T &x) {
  return (T(0) < x) - (x < T(0));
}

bool signbit_(float x) {
  return (*(int*)&x) >> (CHAR_BIT*sizeof(float) - 1);
}

bool signbit_(double x) {
  return (*(long long*)&x) >> (CHAR_BIT*sizeof(double) - 1);
}

template<class Double>
Double copysign_(Double x, Double y) {
  return signbit_(y) ? -fabs(x) : fabs(x);
}

/*

Rounding Functions

- floor0(x) returns x rounded down, symmetrically towards zero.
This function is
  analogous to trunc() in C++11 and later.
- ceil0(x) returns x rounded up, symmetrically away from zero.
This function is
  analogous to round() in C++11 and later.
- round_half_up(x) returns x rounded half up, towards positive
infinity.
- round_half_down(x) returns x rounded half down, towards negative
infinity.
- round_half_to0(x) returns x rounded half down, symmetrically
towards zero.
```

```
- round_half_from0(x) returns x rounded half up, symmetrically
away from zero.
- round_half_even(x) returns x rounded half to even, using
banker's rounding.
- round_half_alternate(x) returns x rounded, where ties are broken
by
  alternating rounds towards positive and negative infinity.
- round_half_alternate0(x) returns x rounded, where ties are
broken by
  alternating symmetric rounds towards and away from zero.
- round_half_random(x) returns x rounded, where ties are broken
randomly.
- round_n_places(x, n, f) returns x rounded to n digits after the
decimal, using
  the specified rounding function f(x).

*/

template<class Double>
Double floor0(const Double &x) {
  Double res = floor(fabs(x));
  return (x < 0.0) ? -res : res;
}

template<class Double>
Double ceil0(const Double &x) {
  Double res = ceil(fabs(x));
  return (x < 0.0) ? -res : res;
}

template<class Double>
Double round_half_up(const Double &x) {
  return floor(x + 0.5);
}

template<class Double>
Double round_half_down(const Double &x) {
  return ceil(x - 0.5);
}

template<class Double>
Double round_half_to0(const Double &x) {
  Double res = round_half_down(fabs(x));
  return (x < 0.0) ? -res : res;
}

template<class Double>
Double round_half_from0(const Double &x) {
  Double res = round_half_up(fabs(x));
  return (x < 0.0) ? -res : res;
}
```

```
template<class Double>
Double round_half_even(const Double &x, const Double &eps = 1e-9)
{
  if (x < 0.0) {
    return -round_half_even(-x, eps);
  }
  Double ipart;
  modf(x, &ipart);
  if (x - (ipart + 0.5) < eps) {
    return (fmod(ipart, 2.0) < eps) ? ipart : ceil0(ipart + 0.5);
  }
  return round_half_from0(x);
}

template<class Double>
Double round_half_alternate(const Double &x) {
  static bool up = true;
  return (up = !up) ? round_half_up(x) : round_half_down(x);
}

template<class Double>
Double round_half_alternate0(const Double &x) {
  static bool up = true;
  return (up = !up) ? round_half_from0(x) : round_half_to0(x);
}

template<class Double>
Double round_half_random(const Double &x) {
  return (rand() % 2 == 0) ? round_half_from0(x) :
round_half_to0(x);
}

template<class Double, class RoundingFunction>
Double round_n_places(const Double &x, unsigned int n,
RoundingFunction f) {
  return f(x*pow(10, n)) / pow(10, n);
}

/*

Error Function

- erf_(x) returns the error encountered in integrating the normal
distribution.
  Its value is 2/sqrt(pi)*(integral of e^(-t^2) dt from 0 to x).
This function
  is analogous to erf(x) in C++11 and later.
- erfc_(x) returns the error function complement, that is, 1 -
erf_(x). This
  function is analogous to erfc(x) in C++11 and later.
```

```
*/

#define ERF_EPS 1e-14

double erfc_(double x);

double erf_(double x) {
  if (signbit_(x)) {
    return -erf_(-x);
  }
  if (fabs(x) > 2.2) {
    return 1.0 - erfc_(x);
  }
  double sum = x, term = x, xx = x*x;
  int j = 1;
  do {
    term *= xx / j;
    sum -= term/(2*(j++) + 1);
    term *= xx / j;
    sum += term/(2*(j++) + 1);
  } while (fabs(term) > sum*ERF_EPS);
  return 2/sqrt(M_PI) * sum;
}

double erfc_(double x) {
  if (fabs(x) < 2.2) {
    return 1.0 - erf_(x);
  }
  if (signbit_(x)) {
    return 2.0 - erfc_(-x);
  }
  double a = 1, b = x, c = x, d = x*x + 0.5, q1, q2 = 0, n = 1.0,
t;
  do {
    t = a*n + b*x;
    a = b;
    b = t;
    t = c*n + d*x;
    c = d;
    d = t;
    n += 0.5;
    q1 = q2;
    q2 = b / d;
  } while (fabs(q1 - q2) > q2*ERF_EPS);
  return 1/sqrt(M_PI) * exp(-x*x) * q2;
}

#undef ERF_EPS

/*
```

```
Gamma Functions

- tgamma_(x) returns the gamma function of x. Unlike the tgamma()
function in
  C++11 and later, this version only supports positive x,
returning NaN if x is
  less than or equal to 0.
- lgamma_(x) returns the natural logarithm of the absolute value
of the gamma
  function of x. Unlike the lgamma() function in C++11 and later,
this version
  only supports positive x, returning NaN if x is less than or
equal to 0.

*/

double lgamma_(double x);

double tgamma_(double x) {
  if (x <= 0) {
    return M_NAN;
  }
  if (x < 1e-3) {
    return 1.0 / (x*(1.0 + 0.5772156649015328606065120 9*x));
  }
  if (x < 12) {
    double y = x;
    int n = 0;
    bool arg_was_less_than_one = (y < 1);
    if (arg_was_less_than_one) {
      y += 1;
    } else {
      n = (int)floor(y) - 1;
      y -= n;
    }
    static const double p[] = {
        -1.71618513886549492533811e+0,
2.47656508055759199108314e+1,
        -3.79804256470945635097577e+2,
6.29331155312818442661052e+2,
        8.66966202790413211295064e+2, -
3.14512729688483675254357e+4,
        -3.61444134186911729807069e+4,
6.64561438202405440627855e+4};
    static const double q[] = {
        -3.08402300119738975254353e+1,
3.15350626979604161529144e+2,
        -1.01515636749021914166146e+3, -
3.10777167157231109440444e+3,
        2.25381184209801510330112e+4,
```

```
4.75584627752788110767815e+3,
        -1.34659959864969306392456e+5, -
1.15132259675553483497211e+5};
    double num = 0, den = 1, z = y - 1;
    for (int i = 0; i < 8; i++) {
      num = (num + p[i])*z;
      den = den*z + q[i];
    }
    double result = num/den + 1;
    if (arg_was_less_than_one) {
      result /= (y - 1);
    } else {
      for (int i = 0; i < n; i++) {
        result *= y++;
      }
    }
    return result;
  }
  return (x > 171.624) ? 2*DBL_MAX : exp(lgamma(x));
}

double lgamma_(double x) {
  if (x <= 0) {
    return M_NAN;
  }
  if (x < 12) {
    return log(fabs(tgamma_(x)));
  }
  static const double c[8] = {
    1.0/12, -1.0/360, 1.0/1260, -1.0/1680, 1.0/1188, -
691.0/360360, 1.0/156,
    -3617.0/122400
  };
  double z = 1.0/(x*x), sum = c[7];
  for (int i = 6; i >= 0; i--) {
    sum = sum*z + c[i];
  }
  return (x - 0.5)*log(x) - x + 0.918938533204672741780329736405662
+ sum/x;
}

/*

Base Conversion

- Given an integer in base a as a vector d of digits (where d[0]
is the least
  significant digit), convert_base(d, a, b) returns a vector of
the integer's
  digits when converted base b (again with index 0 storing the
least significant
```

```
  digit). The actual value of the entire integer to be converted
must be able to
  fit within an unsigned 64-bit integer for intermediate storage.
- convert_digits(x, b) returns the digits of the unsigned integer
x in base b,
  where index 0 of the result stores the least significant digit.
- to_roman(x) returns the Roman numeral representation of the
unsigned integer x
  as a C++ string.

*/

std::vector<int> convert_base(const std::vector<int> &d, int a,
int b) {
  unsigned long long x = 0, power = 1;
  for (int i = 0; i < (int)d.size(); i++) {
    x += d[i]*power;
    power *= a;
  }
  int n = ceil(log(x + 1)/log(b));
  std::vector<int> res;
  for (int i = 0; i < n; i++) {
    res.push_back(x % b);
    x /= b;
  }
  return res;
}

std::vector<int> convert_base(unsigned int x, int b = 10) {
  std::vector<int> res;
  while (x != 0) {
    res.push_back(x % b);
    x /= b;
  }
  return res;
}

std::string to_roman(unsigned int x) {
  static const std::string h[] =
      {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC",
"CM"};
  static const std::string t[] =
      {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX",
"XC"};
  static const std::string o[] =
      {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII",
"IX"};
  std::string prefix(x / 1000, 'M');
  x %= 1000;
  return prefix + h[x/100] + t[x/10 % 10] + o[x % 10];
}
```

```
/*** Example Usage ***/

#include <cassert>
#include <iostream>
int main() {
  assert(EQ(M_PI, 3.14159265359));
  assert(EQ(M_E, 2.718281828459));
  assert(EQ(M_PHI, 1.61803398875));

  double x = -12345.6789;
  assert((-M_INF < x) && (x < M_INF));
  assert((M_INF + x == M_INF) && (M_INF - x == M_INF));
  assert((M_INF + M_INF == M_INF) && (-M_INF - M_INF == -M_INF));
  assert((M_NAN != x) && (M_NAN != M_INF) && (M_NAN != M_NAN));
  assert(!(M_NAN < x) && !(M_NAN > x) && !(M_NAN <= x) && !(M_NAN
>= x));
  assert(isnan(0.0*M_INF) && isnan(0.0*-M_INF) && isnan(M_INF/-
M_INF));
  assert(isnan(M_NAN) && isnan(-M_NAN) && isnan(M_INF - M_INF));

  assert(sgn(x) == -1 && sgn(0.0) == 0 && sgn(5678) == 1);
  assert(signbit_(x) && !signbit_(0.0) && signbit_(-0.0));
  assert(!signbit_(M_INF) && signbit_(-M_INF));
  assert(!signbit_(M_NAN) && signbit_(-M_NAN));
  assert(copysign(1.0, +2.0) == +1.0 && copysign(M_INF, -2.0) == -
M_INF);
  assert(copysign(1.0, -2.0) == -1.0 &&
std::signbit(copysign(M_NAN, -2.0)));

  assert(EQ(floor0(1.5), 1.0) && EQ(ceil0(1.5), 2.0));
  assert(EQ(floor0(-1.5), -1.0) && EQ(ceil0(-1.5), -2.0));
  assert(EQ(round_half_up(+1.5), +2) && EQ(round_half_down(+1.5),
+1));
  assert(EQ(round_half_up(-1.5), -1) && EQ(round_half_down(-1.5),
-2));
  assert(EQ(round_half_to0(+1.5), +1) &&
EQ(round_half_from0(+1.5), +2));
  assert(EQ(round_half_to0(-1.5), -1) && EQ(round_half_from0(-
1.5), -2));
  assert(EQ(round_half_even(+1.5), +2) && EQ(round_half_even(-
1.5), -2));
  assert(NE(round_half_alternate(+1.5),
round_half_alternate(+1.5)));
  assert(NE(round_half_alternate0(-1.5), round_half_alternate0(-
1.5)));
  assert(EQ(round_n_places(-1.23456, 3, round_half_to0<double>), -
1.235));

  assert(EQ(erf_(1.0), 0.8427007929) && EQ(erf_(-1.0), -
0.8427007929));
```

```
  assert(EQ(tgamma_(0.5), 1.7724538509) && EQ(tgamma_(1.0), 1.0));
  assert(EQ(lgamma_(0.5), 0.5723649429) && EQ(lgamma_(1.0), 0.0));

  int digits[] = {6, 5, 4, 3, 2, 1};
  std::vector<int> base20 = convert_base(123456, 20);
  assert(convert_base(base20, 20, 10) == std::vector<int>(digits,
digits + 6));
  assert(to_roman(1234) == "MCCXXXIV");
  assert(to_roman(5678) == "MMMMMDCLXXVIII");
  return 0;
}

/*

The following functions implement common operations in
combinatorics. All input
arguments must be non-negative. All return values and table
entries are computed
as 64-bit integers modulo an input argument m or p.

- factorial(n, m) returns n! mod m.
- factorialp(n, p) returns n! mod p, where p is prime.
- binomial_table(n, m) returns rows 0 to n of Pascal's triangle as
a table t
  such that t[i][j] is equal to (i choose j) mod m.
- permute(n, k, m) returns (n permute k) mod m.
- choose(n, k, p) returns (n choose k) mod p, where p is prime.
- multichoose(n, k, p) returns (n multi-choose k) mod p, where p
is prime.
- catalan(n, p) returns the nth Catalan number mod p, where p is
prime.
- partitions(n, m) returns the number of partitions of n, mod m.
- partitions(n, k, m) returns the number of partitions of n into k
parts, mod m.
- stirling1(n, k, m) returns the (n, k) unsigned Stirling number
of the 1st kind
  mod m.
- stirling2(n, k, m) returns the (n, k) Stirling number of the 2nd
kind mod m.
- eulerian1(n, k, m) returns the (n, k) Eulerian number of the 1st
kind mod m,
  where n > k.
- eulerian2(n, k, m) returns the (n, k) Eulerian number of the 2nd
kind mod m,
  where n > k.

Time Complexity:
- O(n) for factorial(n, m).
- O(p log n) for factorialp(n, p).
- O(n^2) for binomial_table(n, m).
- O(k) for permute(n, k, p).
```

```
- O(min(k, n - k)) for choose(n, k, p).
- O(k) for multichoose(n, k, p).
- O(n) for catalan(n, p).
- O(n^2) for partitions(n, m).
- O(n*k) for partitions(n, k, m), stirling1(n, k, m), stirling2(n,
k, m),
  eulerian1(n, k, m), and eulerian2(n, k, m).

Space Complexity:
- O(n^2) auxiliary heap space for binomial_table(n, m).
- O(n*k) auxiliary heap space for partitions(n, k, m),
stirling1(n, k, m),
  stirling2(n, k, m), eulerian1(n, k, m), and eulerian2(n, k, m).
- O(1) auxiliary for all other operations.

*/

#include <vector>

typedef long long int64;
typedef std::vector<std::vector<int64> > table;

int64 factorial(int n, int m = 1000000007) {
  int64 res = 1;
  for (int i = 2; i <= n; i++) {
    res = (res*i) % m;
  }
  return res % m;
}

int64 factorialp(int64 n, int64 p = 1000000007) {
  int64 res = 1;
  while (n > 1) {
    if (n / p % 2 == 1) {
      res = res*(p - 1) % p;
    }
    int max = n % p;
    for (int i = 2; i <= max; i++) {
      res = (res*i) % p;
    }
    n /= p;
  }
  return res % p;
}

table binomial_table(int n, int64 m = 1000000007) {
  table t(n + 1);
  for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= i; j++) {
      if (i < 2 || j == 0 || i == j) {
        t[i].push_back(1);
```

```
      } else {
        t[i].push_back((t[i - 1][j - 1] + t[i - 1][j]) % m);
      }
    }
  }
  return t;
}

int64 permute(int n, int k, int64 m = 1000000007) {
  if (n < k) {
    return 0;
  }
  int64 res = 1;
  for (int i = 0; i < k; i++) {
    res = res*(n - i) % m;
  }
  return res % m;
}

int64 mulmod(int64 x, int64 n, int64 m) {
  int64 a = 0, b = x % m;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = (a + b) % m;
    }
    b = (b << 1) % m;
  }
  return a % m;
}

int64 powmod(int64 x, int64 n, int64 m) {
  int64 a = 1, b = x;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = mulmod(a, b, m);
    }
    b = mulmod(b, b, m);
  }
  return a % m;
}

int64 choose(int n, int k, int64 p = 1000000007) {
  if (n < k) {
    return 0;
  }
  if (k > n - k) {
    k = n - k;
  }
  int64 num = 1, den = 1;
  for (int i = 0; i < k; i++) {
    num = num*(n - i) % p;
```

```
  }
  for (int i = 1; i <= k; i++) {
    den = den*i % p;
  }
  return num*powmod(den, p - 2, p) % p;
}

int64 multichoose(int n, int k, int64 p = 1000000007) {
  return choose(n + k - 1, k, p);
}

int64 catalan(int n, int64 p = 1000000007) {
  return choose(2*n, n, p)*powmod(n + 1, p - 2, p) % p;
}

int64 partitions(int n, int64 m = 1000000007) {
  std::vector<int64> t(n + 1, 0);
  t[0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
      t[j] = (t[j] + t[j - i]) % m;
    }
  }
  return t[n] % m;
}

int64 partitions(int n, int k, int64 m = 1000000007) {
  table t(n + 1, std::vector<int64>(k + 1, 0));
  t[0][1] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1, h = k < i ? k : i; j <= h; j++) {
      t[i][j] = (t[i - 1][j - 1] + t[i - j][j]) % m;
    }
  }
  return t[n][k] % m;
}

int64 stirling1(int n, int k, int64 m = 1000000007) {
  table t(n + 1, std::vector<int64>(k + 1, 0));
  t[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      t[i][j] = (i - 1)*t[i - 1][j] % m;
      t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
    }
  }
  return t[n][k] % m;
}

int64 stirling2(int n, int k, int64 m = 1000000007) {
  table t(n + 1, std::vector<int64>(k + 1, 0));
```

```
  t[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      t[i][j] = j*t[i - 1][j] % m;
      t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
    }
  }
  return t[n][k] % m;
}

int64 eulerian1(int n, int k, int64 m = 1000000007) {
  if (k > n - 1 - k) {
    k = n - 1 - k;
  }
  table t(n + 1, std::vector<int64>(k + 1, 1));
  for (int j = 1; j <= k; j++) {
    t[0][j] = 0;
  }
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      t[i][j] = (i - j)*t[i - 1][j - 1] % m;
      t[i][j] = (t[i][j] + ((j + 1)*t[i - 1][j] % m)) % m;
    }
  }
  return t[n][k] % m;
}

int64 eulerian2(int n, int k, int64 m = 1000000007) {
  table t(n + 1, std::vector<int64>(k + 1, 1));
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      if (i == j) {
        t[i][j] = 0;
      } else {
        t[i][j] = (j + 1)*t[i - 1][j] % m;
        t[i][j] = ((2*i - 1 - j)*t[i - 1][j - 1] % m + t[i][j]) %
m;
      }
    }
  }
  return t[n][k] % m;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  table t = binomial_table(10);
  for (int i = 0; i < (int)t.size(); i++) {
    for (int j = 0; j < (int)t[i].size(); j++) {
```

```
      assert(t[i][j] == choose(i, j));
    }
  }
  assert(factorial(10) == 3628800);
  assert(factorialp(123456) == 639390503);
  assert(permute(10, 4) == 5040);
  assert(choose(20, 7) == 77520);
  assert(multichoose(20, 7) == 657800);
  assert(catalan(10) == 16796);
  assert(partitions(4) == 5);
  assert(partitions(100, 5) == 38225);
  assert(stirling1(4, 2) == 11);
  assert(stirling2(4, 3) == 6);
  assert(eulerian1(9, 5) == 88234);
  assert(eulerian2(8, 3) == 195800);
  return 0;
}

/*

Generate prime numbers using the Sieve of Eratosthenes.

- sieve(n) returns a vector of all the primes less than or equal
to n.
- sieve(lo, hi) returns a vector of all the primes in the range
[lo, hi].

Time Complexity:
- O(n log(log(n))) per call to sieve(n).
- O(sqrt(hi)*log(log(hi - lo))) per call to sieve(lo, hi).

Space Complexity:
- O(n) auxiliary heap space per call to sieve(n).
- O(hi - lo + sqrt(hi)) auxiliary heap space per call to sieve(lo,
hi).

*/

#include <cmath>
#include <vector>

std::vector<int> sieve(int n) {
  std::vector<bool> prime(n + 1, true);
  int sqrtn = ceil(sqrt(n));
  for (int i = 2; i <= sqrtn; i++) {
    if (prime[i]) {
      for (int j = i*i; j <= n; j += i) {
        prime[j] = false;
      }
    }
  }
```

```
  std::vector<int> res;
  for (int i = 2; i <= n; i++) {
    if (prime[i]) {
      res.push_back(i);
    }
  }
  return res;
}

std::vector<int> sieve(int lo, int hi) {
  int sqrt_hi = ceil(sqrt(hi)), fourth_root_hi =
ceil(sqrt(sqrt_hi));
  std::vector<bool> prime1(sqrt_hi + 1, true), prime2(hi - lo + 1,
true);
  for (int i = 2; i <= fourth_root_hi; i++) {
    if (prime1[i]) {
      for (int j = i*i; j <= sqrt_hi; j += i) {
        prime1[j] = false;
      }
    }
  }
  for (int i = 2, n = hi - lo; i <= sqrt_hi; i++) {
    if (prime1[i]) {
      for (int j = (lo / i)*i - lo; j <= n; j += i) {
        if (j >= 0 && j + lo != i) {
          prime2[j] = false;
        }
      }
    }
  }
  std::vector<int> res;
  for (int i = (lo > 1) ? lo : 2; i <= hi; i++) {
    if (prime2[i - lo]) {
      res.push_back(i);
    }
  }
  return res;
}

/*** Example Usage and Output:

sieve(n=10000000): 0.059s
atkins(n=10000000): 0.08s
sieve([1000000000, 1005000000]): 0.034s

***/

#include <ctime>
#include <iostream>
using namespace std;
```

```
int main() {
  int pmax = 10000000;
  vector<int> p;
  time_t start;
  double delta;

  start = clock();
  p = sieve(pmax);
  delta = (double)(clock() - start)/CLOCKS_PER_SEC;
  cout << "sieve(n=" << pmax << "): " << delta << "s" << endl;

  int l = 1000000000, h = 1005000000;
  start = clock();
  p = sieve(l, h);
  delta = (double)(clock() - start)/CLOCKS_PER_SEC;
  cout << "sieve([" << l << ", " << h << "]): " << delta << "s" <<
endl;
  return 0;
}

/*

Determine whether an integer n is prime. This can be done
deterministically by
testing all numbers under sqrt(n) using trial division,
probabilistically using
the Miller-Rabin test, or deterministically using the Miller-Rabin
test if the
maximum input is known (2^63 - 1 for the purposes here).

- is_prime(n) returns whether the integer n is prime using an
optimized trial
  division technique based on the fact that all primes greater
than 6 must take
  the form 6n + 1 or 6n - 1.
- is_probable_prime(n, k) returns true if the integer n is prime,
or false with
  an error probability of (1/4)^k if n is composite. In other
words, the result
  is guaranteed to be correct if n is prime, but could be wrong
with probability
  (1/4)^k if n is composite. This implementation uses uses
exponentiation by
  squaring to support all signed 64-bit integers (up to and
including 2^63 - 1).
- is_prime_fast(n) returns whether the signed 64-bit integer n is
prime using
  a fully deterministic version of the Miller-Rabin test.

Time Complexity:
- O(sqrt n) per call to is_prime(n).
```

```
- O(k log^3(n)) per call to is_probable_prime(n, k).
- O(log^3(n)) per call to is_prime_fast(n).

Space Complexity:
- O(1) auxiliary space for all operations.

*/

#include <cstdlib>

template<class Int>
bool is_prime(Int n) {
  if (n == 2 || n == 3) {
    return true;
  }
  if (n < 2 || n % 2 == 0 || n % 3 == 0) {
    return false;
  }
  for (Int i = 5, w = 4; i*i <= n; i += w) {
    if (n % i == 0) {
      return false;
    }
    w = 6 - w;
  }
  return true;
}

typedef unsigned long long uint64;

uint64 mulmod(uint64 x, uint64 n, uint64 m) {
  uint64 a = 0, b = x % m;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = (a + b) % m;
    }
    b = (b << 1) % m;
  }
  return a % m;
}

uint64 powmod(uint64 x, uint64 n, uint64 m) {
  uint64 a = 1, b = x;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = mulmod(a, b, m);
    }
    b = mulmod(b, b, m);
  }
  return a % m;
}
```

```
uint64 rand64u() {
  return ((uint64)(rand() & 0xf) << 60) |
         ((uint64)(rand() & 0x7fff) << 45) |
         ((uint64)(rand() & 0x7fff) << 30) |
         ((uint64)(rand() & 0x7fff) << 15) |
         ((uint64)(rand() & 0x7fff));
}

bool is_probable_prime(long long n, int k = 20) {
  if (n == 2 || n == 3) {
    return true;
  }
  if (n < 2 || n % 2 == 0 || n % 3 == 0) {
    return false;
  }
  uint64 s = n - 1, p = n - 1;
  while (!(s & 1)) {
    s >>= 1;
  }
  for (int i = 0; i < k; i++) {
    uint64 x, r = powmod(rand64u() % p + 1, s, n);
    for (x = s; x != p && r != 1 && r != p; x <<= 1) {
      r = mulmod(r, r, n);
    }
    if (r != p && !(x & 1)) {
      return false;
    }
  }
  return true;
}

bool is_prime_fast(long long n) {
  static const int np = 9, p[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  for (int i = 0; i < np; i++) {
    if (n % p[i] == 0) {
      return n == p[i];
    }
  }
  if (n < p[np - 1]) {
    return false;
  }
  uint64 t;
  int s = 0;
  for (t = n - 1; !(t & 1); t >>= 1) {
    s++;
  }
  for (int i = 0; i < np; i++) {
    uint64 r = powmod(p[i], t, n);
    if (r == 1) {
      continue;
    }
    bool ok = false;
```

```
    for (int j = 0; j < s && !ok; j++) {
      ok |= (r == (uint64)n - 1);
      r = mulmod(r, r, n);
    }
    if (!ok) {
      return false;
    }
  }
  return true;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  int len = 20;
  long long tests[] = {
    -1, 0, 1, 2, 3, 4, 5, 1000000LL, 772023803LL, 792904103LL,
813815117LL,
    834753187LL, 855718739LL, 876717799LL, 897746119LL,
2147483647LL,
    5705234089LL, 5914686649LL, 6114145249LL, 6339503641LL,
6548531929LL
  };
  for (int i = 0; i < len; i++) {
    bool p = is_prime(tests[i]);
    assert(p == is_prime_fast(tests[i]));
    assert(p == is_probable_prime(tests[i]));
  }
  return 0;
}

/*

Useful or trivial string operations. These functions are not
particularly
algorithmic. They are typically naive implementations using C++
features.
They depend on many features of the C++ <string> library, which
tend to
have an unspecified complexity. They may not be optimally
efficient.

*/

#include <cstdlib>
#include <sstream>
#include <string>
#include <vector>
```

```
//integer to string conversion and vice versa using C++ features

//note that a similar std::to_string is introduced in C++0x
template<class Int>
std::string to_string(const Int & i) {
  std::ostringstream oss;
  oss << i;
  return oss.str();
}

//like atoi, except during special cases like overflows
int to_int(const std::string & s) {
  std::istringstream iss(s);
  int res;
  if (!(iss >> res)) /* complain */;
  return res;
}

/*

itoa implementation (fast)
documentation: http://www.cplusplus.com/reference/cstdlib/itoa/
taken from: http://www.jb.man.ac.uk/~slowe/cpp/itoa.html

*/

char* itoa(int value, char * str, int base = 10) {
  if (base < 2 || base > 36) {
    *str = '\0';
    return str;
  }
  char *ptr = str, *ptr1 = str, tmp_c;
  int tmp_v;
  do {
    tmp_v = value;
    value /= base;
    *ptr++ = "zyxwvutsrqponmlkjihgfedcba9876543210123456789"
             "abcdefghijklmnopqrstuvwxyz"[35 + (tmp_v - value *
base)];
  } while (value);
  if (tmp_v < 0) *ptr++ = '-';
  for (*ptr-- = '\0'; ptr1 < ptr; *ptr1++ = tmp_c) {
    tmp_c = *ptr;
    *ptr-- = *ptr1;
  }
  return str;
}

/*

Trimming functions (in place). Given a string and optionally a
```

```
series
of characters to be considered for trimming, trims the string's
ends
(left, right, or both) and returns the string. Note that the
ORIGINAL
string is trimmed as it's passed by reference, despite the
original
reference being returned for convenience.

*/

std::string& ltrim(std::string & s, const std::string & delim = "
\n\t\v\f\r") {
  unsigned int pos = s.find_first_not_of(delim);
  if (pos != std::string::npos) s.erase(0, pos);
  return s;
}

std::string& rtrim(std::string & s, const std::string & delim = "
\n\t\v\f\r") {
  unsigned int pos = s.find_last_not_of(delim);
  if (pos != std::string::npos) s.erase(pos);
  return s;
}

std::string& trim(std::string & s, const std::string & delim = "
\n\t\v\f\r") {
  return ltrim(rtrim(s));
}

/*

Returns a copy of the string s with all occurrences of the given
string search replaced with the given string replace.

Time Complexity: Unspecified, but proportional to the number of
times
the search string occurs and the complexity of
std::string::replace,
which is unspecified.

*/

std::string replace(std::string s,
                    const std::string & search,
                    const std::string & replace) {
  if (search.empty()) return s;
  unsigned int pos = 0;
  while ((pos = s.find(search, pos)) != std::string::npos) {
    s.replace(pos, search.length(), replace);
    pos += replace.length();
```

```
  }
  return s;
}

/*

Tokenizes the string s based on single character delimiters.

Version 1: Simpler. Only one delimiter character allowed, and this
will
not skip empty tokens.
  e.g. split("a::b", ":") yields {"a", "b"}, not {"a", "", "b"}.

Version 2: All of the characters in the delim parameter that also
exists
in s will be removed from s, and the token(s) of s that are left
over will
be added sequentially to a vector and returned. Empty tokens are
skipped.
  e.g. split("a::b", ":") yields {"a", "b"}, not {"a", "", "b"}.

Time Complexity: O(s.length() * delim.length())

*/

std::vector<std::string> split(const std::string & s, char delim)
{
  std::vector<std::string> res;
  std::stringstream ss(s);
  std::string curr;
  while (std::getline(ss, curr, delim))
    res.push_back(curr);
  return res;
}

std::vector<std::string> split(const std::string & s,
                               const std::string & delim = "
\n\t\v\f\r") {
  std::vector<std::string> res;
  std::string curr;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim.find(s[i]) == std::string::npos) {
      curr += s[i];
    } else if (!curr.empty()) {
      res.push_back(curr);
      curr = "";
    }
  }
  if (!curr.empty()) res.push_back(curr);
  return res;
}
```

```
/*

Like the explode() function in PHP, the string s is tokenized
based
on delim, which is considered as a whole boundary string, not just
a
sequence of possible boundary characters like the split() function
above.
This will not skip empty tokens.
  e.g. explode("a::b", ":") yields {"a", "", "b"}, not {"a", "b"}.

Time Complexity: O(s.length() * delim.length())

*/

std::vector<std::string> explode(const std::string & s,
                                 const std::string & delim) {
  std::vector<std::string> res;
  unsigned int last = 0, next = 0;
  while ((next = s.find(delim, last)) != std::string::npos) {
    res.push_back(s.substr(last, next - last));
    last = next + delim.size();
  }
  res.push_back(s.substr(last));
  return res;
}

/*** Example Usage ***/

#include <cassert>
#include <cstdio>
#include <iostream>
using namespace std;

void print(const vector<string> & v) {
  cout << "[";
  for (int i = 0; i < (int)v.size(); i++)
    cout << (i ? "\", \"" : "\"") << v[i];
  cout << "\"]\n";
}

int main() {
  assert(to_string(123) + "4" == "1234");
  assert(to_int("1234") == 1234);
  char buffer[50];
  assert(string(itoa(1750, buffer, 10)) == "1750");
  assert(string(itoa(1750, buffer, 16)) == "6d6");
  assert(string(itoa(1750, buffer, 2)) == "11011010110");

  string s("   abc \n");
```

```
  string t = s;
  assert(ltrim(s) == "abc \n");
  assert(rtrim(s) == trim(t));
  assert(replace("abcdabba", "ab", "00") == "00cd00ba");

  vector<string> tokens;

  tokens = split("a\nb\ncde\nf", '\n');
  cout << "split v1: ";
  print(tokens); //["a", "b", "cde", "f"]

  tokens = split("a::b,cde:,f", ":,");
  cout << "split v2: ";
  print(tokens); //["a", "b", "cde", "f"]

  tokens = explode("a..b.cde....f", "..");
  cout << "explode: ";
  print(tokens); //["a", ".b.cde", "", ".f"]
  return 0;
}

/*

Evaluate a mathematica expression in accordance to the order
of operations (parentheses, exponents, multiplication, division,
addition, subtraction). This handles unary operators like '-'.

*/

#include <string>

template<class It> int eval(It & it, int prec) {
  if (prec == 0) {
    int sign = 1, ret = 0;
    for (; *it == '-'; it++) sign *= -1;
    if (*it == '(') {
      ret = eval(++it, 2);
      it++;
    } else while (*it >= '0' && *it <= '9') {
      ret = 10 * ret + (*(it++) - '0');
    }
    return sign * ret;
  }
  int num = eval(it, prec - 1);
  while (!((prec == 2 && *it != '+' && *it != '-') ||
           (prec == 1 && *it != '*' && *it != '/'))) {
    switch (*(it++)) {
      case '+': num += eval(it, prec - 1); break;
      case '-': num -= eval(it, prec - 1); break;
      case '*': num *= eval(it, prec - 1); break;
      case '/': num /= eval(it, prec - 1); break;
```

```
    }
  }
  return num;
}

/*** Wrapper Function ***/

int eval(const std::string & s) {
  std::string::iterator it = std::string(s).begin();
  return eval(it, 2);
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
  cout << eval("1+2*3*4+3*(2+2)-100") << "\n";
  return 0;
}

/*

Given an text and a pattern to be searched for within the text,
determine the first position in which the pattern occurs in
the text. The KMP algorithm is much faster than the naive,
quadratic time, string searching algorithm that is found in
string.find() in the C++ standard library.

KMP generates a table using a prefix function of the pattern.
Then, the precomputed table of the pattern can be used
indefinitely
for any number of texts.

Time Complexity: O(n + m) where n is the length of the text
and m is the length of the pattern.

Space Complexity: O(m) auxiliary on the length of the pattern.

*/

#include <string>
#include <vector>

int find(const std::string & text, const std::string & pattern) {
  if (pattern.empty()) return 0;
  //generate table using pattern
  std::vector<int> p(pattern.size());
  for (int i = 0, j = p[0] = -1; i < (int)pattern.size(); ) {
    while (j >= 0 && pattern[i] != pattern[j])
```

```
      j = p[j];
    i++;
    j++;
    p[i] = (pattern[i] == pattern[j]) ? p[j] : j;
  }
  //use the precomputed table to search within text
  //the following can be repeated on many different texts
  for (int i = 0, j = 0; j < (int)text.size(); ) {
    while (i >= 0 && pattern[i] != text[j])
      i = p[i];
    i++;
    j++;
    if (i >= (int)pattern.size())
      return j - i;
  }
  return std::string::npos;
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(15 == find("ABC ABCDAB ABCDABCDABDE", "ABCDABD"));
  return 0;
}


/*

A substring is a consecutive part of a longer string (e.g. "ABC"
is
a substring of "ABCDE" but "ABD" is not). Using dynamic
programming,
determine the longest string which is a substring common to any
two
input strings.

Time Complexity: O(n * m) where n and m are the lengths of the two
input strings, respectively.

Space Complexity: O(min(n, m)) auxiliary.

*/

#include <string>

std::string longest_common_substring
(const std::string & s1, const std::string & s2) {
  if (s1.empty() || s2.empty()) return "";
  if (s1.size() < s2.size())
```

```
    return longest_common_substring(s2, s1);
  int * A = new int[s2.size()];
  int * B = new int[s2.size()];
  int startpos = 0, maxlen = 0;
  for (int i = 0; i < (int)s1.size(); i++) {
    for (int j = 0; j < (int)s2.size(); j++) {
      if (s1[i] == s2[j]) {
        A[j] = (i > 0 && j > 0) ? 1 + B[j - 1] : 1;
        if (maxlen < A[j]) {
          maxlen = A[j];
          startpos = i - A[j] + 1;
        }
      } else {
        A[j] = 0;
      }
    }
    int * temp = A;
    A = B;
    B = temp;
  }
  delete[] A;
  delete[] B;
  return s1.substr(startpos, maxlen);
}

/*** Example Usage ***/

#include <cassert>

int main() {
  assert(longest_common_substring("bbbabca", "aababcd") ==
"babc");
  return 0;
}

/*

A subsequence is a sequence that can be derived from another
sequence
by deleting some elements without changing the order of the
remaining
elements (e.g. "ACE" is a subsequence of "ABCDE", but "BAE" is
not).
Using dynamic programming, determine the longest string which
is a subsequence common to any two input strings.

In addition, the shortest common supersequence between two strings
is
a closely related problem, which involves finding the shortest
string
which has both input strings as subsequences (e.g. "ABBC" and
```

```
"BCB" has
the shortest common supersequence of "ABBCB"). The answer is
simply:
  (sum of lengths of s1 and s2) - (length of LCS of s1 and s2)

Time Complexity: O(n * m) where n and m are the lengths of the two
input strings, respectively.

Space Complexity: O(n * m) auxiliary.

*/

#include <string>
#include <vector>

std::string longest_common_subsequence
(const std::string & s1, const std::string & s2) {
  int n = s1.size(), m = s2.size();
  std::vector< std::vector<int> > dp;
  dp.resize(n + 1, std::vector<int>(m + 1, 0));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (s1[i] == s2[j]) {
        dp[i + 1][j + 1] = dp[i][j] + 1;
      } else if (dp[i + 1][j] > dp[i][j + 1]) {
        dp[i + 1][j + 1] = dp[i + 1][j];
      } else {
        dp[i + 1][j + 1] = dp[i][j + 1];
      }
    }
  }
  std::string ret;
  for (int i = n, j = m; i > 0 && j > 0; ) {
    if (s1[i - 1] == s2[j - 1]) {
      ret = s1[i - 1] + ret;
      i--;
      j--;
    } else if (dp[i - 1][j] < dp[i][j - 1]) {
      j--;
    } else {
      i--;
    }
  }
  return ret;
}

/*** Example Usage ***/

#include <cassert>

int main() {
```

```
  assert(longest_common_subsequence("xmjyauz", "mzjawxu") ==
"mjau");
  return 0;
}
```