```
// Dijkstra

#include <bits/stdc++.h>
#define FR(i, n) for (int i = 0; i < (n); ++i)
using namespace std;

typedef vector<int> vi;
typedef pair<int, int> ii;

vector< vector<ii> > g;
vi D;
int n;

void dijkstra(int s) {
   D.assign(n, -1);

   priority_queue<ii, vector<ii>, greater<ii> > pq;
   pq.emplace(0, s);

   while (!pq.empty()) {
      int u = pq.top().second, d = pq.top().first; pq.pop();

      if (D[u] != -1) continue;
      D[u] = d;

      for (auto &i : g[u]) {
     int v = i.second, w = i.first;
     pq.emplace(d + w, v);
      }
   }
}

int main() {

   cin >> n;
   g.assign(n, vector<ii>());

   // for each edge
   int m = 5;
   FR(i, m) {
      int u, v, w;
      cin >> u >> v >> w;
      g[u].push_back(ii(w, v));
      g[v].push_back(ii(w, u));
   }

   // run dijkstra from some src
   dijkstra(0);

   // path? for every edge from u -> v, if D[v] == D[u] + weight(u, v),
   // then u -> v is an edge in "some" optimal path.

}


// ================================================================================

class RMQ {
public:

   vector<int> A;
   vector< vector<int> > M;
   RMQ(const vector<int> &B) {
      A = B;
      int n = A.size();
      int m = 31 - __builtin_clz(n) + 1;
```

```
      M.assign(m, vector<int>(n));
      for (int j = 0; j < n; j++) M[0][j] = j;

      for (int i = 1; (1 << i) <= n; i++) {
     for (int j = 0; (j + (1 << i)) <= n; j++) {
        M[i][j] = (A[M[i - 1][j]] <= A[M[i - 1][j + (1 << (i - 1))]])
           ? M[i - 1][j]
           : M[i - 1][j + (1 << (i - 1))];
     }
      }
   }

   int query(int L, int R) {
      int k = 31 - __builtin_clz(R - L + 1);
      return (A[M[k][L]] <= A[M[k][R - (1 << k) + 1]])
     ? M[k][L]
     : M[k][R - (1 << k) + 1];
   }

};

// ================================================================================

// LIS/LDS/DP O(n^2)

// DP N^2 to find and store length of longest ascending subsequence
for (int i = n - 1; i >= 0; i--) {
   asc[i] = 1;
   for (int j = i + 1; j < n; j++) {
      // less than, for ascending subseq
      if (A[i] < A[j]) {
     asc[i] = max(asc[i], asc[j] + 1);
      }
   }
}

for (int i = n - 1; i >= 0; i--) {
   desc[i] = 1;
   for (int j = i + 1; j < n; j++) {
      // greater than, for descending subseq
      if (A[i] > A[j]) {
     desc[i] = max(desc[i], desc[j] + 1);
      }
   }
}

// ================================================================================

// Longest ascending subsequence O(n log n)

vi LIS(vi &A) {
  int n = (int)A.size();

  // Uncomment to find for descending subsequence
  // reverse(A.begin(), A.end());

  vi last(n + 1), pos(n + 1), pred(n);
  if (n == 0)
    return vi();

  int len = 1;
  last[1] = A[pos[1] = 0];

  for (int i = 1; i < n; i++) {
    // upper_bound for ascending
```

```
    int j = upper_bound(last.begin() + 1, last.begin() + len + 1, A[i]) -
            last.begin();
    pred[i] = (j - 1 > 0) ? pos[j - 1] : -1;
    last[j] = A[pos[j] = i];
    len = max(len, j);
  }

  int start = pos[len];
  vi S(len);
  for (int i = len - 1; i >= 0; i--) {
    S[i] = A[start];
    start = pred[start];
  }

  return S;
}

vi LIS_strict(vi &A) {
  int n = (int)A.size();

  // Uncomment to find for descending subsequence
  // reverse(A.begin(), A.end());

  vi last(n + 1), pos(n + 1), pred(n);
  if (n == 0)
    return vi();

  int len = 1;
  last[1] = A[pos[1] = 0];

  for (int i = 1; i < n; i++) {
    // lower_bound for strictly ascending
    int j = lower_bound(last.begin() + 1, last.begin() + len + 1, A[i]) -
            last.begin();
    pred[i] = (j - 1 > 0) ? pos[j - 1] : -1;
    last[j] = A[pos[j] = i];
    len = max(len, j);
  }

  int start = pos[len];
  vi S(len);
  for (int i = len - 1; i >= 0; i--) {
    S[i] = A[start];
    start = pred[start];
  }

  return S;
}

// ============================================================================

// Tarjan's SCC

#define UNVISITED -1

int depth, num_scc;
vi num, lo, stk, vis;
vvi g;

// Tarjan's algorithm to find SCC's of directed graph; O(V+E).
// --> commonly used for pre-processing to contract digraph to DAG
void scc(int u) {
  lo[u] = num[u] = depth++; // lo[u] <= num[u]

  // push 'u' onto stack, and track explored vertices with 'vis'
  stk.push_back(u);
```

```
  vis[u] = 1;
  for (auto &v : g[u]) {
    if (num[v] == UNVISITED)
      scc(v); // this part is amortized O(V)

    // Condition to update:
    if (vis[v])
      lo[u] = min(lo[u], lo[v]);
  }

  // if root, i.e. start of an SCC
  // Since only visited vertices may update lo[u], and initally we set lo[u] =
  // num[u], then if lo[u] == num[u], we know 'u' is the root of this SCC.  To
  // access the members of this SCC, pop from our "stack" (i.e. 'stk', as
  // vector), up to (and including) root 'u'.
  if (lo[u] == num[u]) {
    cout << "SCC #" << ++num_scc << ":";
    for (;;) {
      int v = stk.back();
      stk.pop_back();
      vis[v] = 0;
      cout << " \n"[u == v] << v;
      if (u == v)
        break;
    }
  }
}

int main() {

  // Number of vertices
  int V;
  V = 10;

  // Build adjacency list
  g.assign(V, vi());
  // ...

  depth = num_scc = 0;
  num.assign(V, UNVISITED);
  lo.assign(V, 0);
  vis.assign(V, 0);
  REP(i, V) {
    if (num[i] == UNVISITED) {
      scc(i);
    }
  }

  return 0;
}


// ============================================================================

// Find articulation points and bridges


// doesn't matter the value, just different
#define UNVISITED -1

// need these for APB algorithm, initialized in main()
int depth, root, children;
vi lo, num, parent;

// true for index i if i identifies an articulation point
vector<bool> art_points;
```

```
// adj list representation of graph
vector<vi> adj_list;

// Find Articulation Points and Bridges
void APB(int u) {
  lo[u] = num[u] = depth++; // lo[u] <= num[u]
  for (auto &v : adj_list[u]) {

    if (num[v] == UNVISITED) {

      // tree edge
      parent[v] = u;
      if (u == root)
        children++;

      // Recurse
      APB(v);

      // articulation point
      if (lo[v] >= num[u])
        art_points[u] = true;

      // bridge
      if (lo[v] > num[u])
        cout << "Edge " << u << " -> " << v << " is a bridge\n";

      // update lo
      lo[u] = min(lo[u], lo[v]);

    } else if (v != parent[u]) {
      // back edge, not a direct cycle
      lo[u] = min(lo[u], num[v]);
    }
  }
}

void SolveForArticulationPointsAndBridges() {
  // setup adjlist
  int V;
  V = 10;

  // find articulation points and bridges
  depth = 0;
  num.assign(V, UNVISITED);
  lo.assign(V, 0);
  parent.assign(V, -1);
  art_points.assign(V, false);

  // first find bridges
  cout << "Bridges:\n";
  FOR(i, V) {
    if (num[i] == UNVISITED) {
      root = i, children = 0;
      APB(i);
      art_points[root] = (children > 1);
    }
  }

  // points
  cout << "Points:\n";
  FOR(i, V) {
    if (art_points[i])
      cout << "Vertex " << i << endl;
  }
}
```

```


// =============================================================================

// Edmonds Karp - Max flow, O(V^3 * E) using adj mat


#define INF 0x3f3f3f3f

// mf: max flow (our solution, eventually)
// f : min f at the time
// s : source
// t : sink
int mf, f, s, t;

// parent, for BFS
vi p;

// max # of vertices we could have
#define MV 1000
int res[MV][MV];

void augment(int v, int lo) {
  if (v == s) {
    // base case: we reach source, and resolve 'f' to the minimum edge
    f = lo;
    return;
  } else if (p[v] != -1) {
    // wonder... is this check redundant? ==> no. It is needed because not
    // necessarily have v -> u (back) for all u -> v

    // recursive call
    augment(p[v], min(lo, res[p[v]][v]));

    // NOTE: changing with 'f', not 'lo'... why?
    // because we're doing recursive calls, updating 'f' once we arrive at
    // 's', at which point the global 'f' is updated, 'augment' returns up the
    // stack, and these values need to be updated with that 'f', not just what
    // 'lo' was at the time
    //
    // update forward res edges u -> v (i.e. p[v] -> v) with -f
    // update backwrd res edges v -> u (i.e. v -> p[v]) with +f
    res[p[v]][v] -= f;
    res[v][p[v]] += f;
  }
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  // for each case, setup res, s, t
  cin >> s >> t;

  // init to 0, need to do this cause using adj matrix
  REP(i, MV)
  REP(j, MV) res[i][j] = 0;

  // of the form:
  // node_u node_v uv_capacity
  int u, v, c;
  while (cin >> u >> v >> c) {
    cin >> u >> v >> c;
    res[u][v] = c;
  }

```

```
  // max flow, initially 0
  mf = 0;
  for (;;) {
    // don't forget this.  'f' needs to be reset each time
    // or does it?... don't see why
    // ==> YES: it is needed,
    // ==> See below: <f needs to be 0>
    f = 0;

    // here, dist is the 'hop-count' from s to t
    // but the actual 'weights' (i.e. capacities) of the edges are
    // handled in 'res', initially equal to original values, but
    // change over the course of the algorithm
    vi dist(MV, INF);
    dist[s] = 0;

    // don't forget this part
    p.assign(MV, -1);

    queue<int> q;
    q.push(s);
    while (!q.empty()) {
      auto u = q.front();
      q.pop();
      if (u == t) break;  // Exit condition: reach the sink

      REP(v, MV) {
        // Interpretation
        // 1: res[u][v] > 0
        // ==> v reachable from u
        // 2: dist[v] == INF
        // ==> unvisited
        if (res[u][v] > 0 && dist[v] == INF)
          dist[v] = dist[u] + 1, q.push(u), p[v] = u;
      }
    }
    // BFS path has been created and stored by 'p', now augment.
    // When this completes, we will know the minimum flow 'f' in this path ***IF
    // ANY*** why IF ANY?  ==> because when 'f' == 0, we cannot send any more
    // flow, so we are DONE
    // ==> See above: <f needs to be 0>
    augment(t, INF);
    if (f == 0) break;

    // we increase 'mf' maxflow solution for this case, by 'f' -> the flow that
    // we still have the capacity to send.
    mf += f;
  }

  // print 'mf' the maximum flow.
  cout << mf << endl;

  return 0;
}


// ============================================================================

// Find tree diameter

// graph representation of our tree, as an adjacency list
vvi g;

// Returns a pair<int, int>, first is the greatest depth found
// from 's', second is the node ID where that depth was found.
ii dfs(int s, int e) {
```

```
  ii b = ii(0, s);
  for (auto &u : g[s]) {

    // only recurse on children (i.e. below 'u')
    if (u != e) {
      ii x = dfs(u, s);
      x.L++;

      // update, if greatest depth seen so far
      if (x.L > b.L)
        b = x;
    }
  }
  return b;
}

void tree_diameter() {
  ii b = dfs(0, 0);
  ii c = dfs(b.R, b.R);
  cout << "diameter is " << c.L << ", found between nodes " << b.R << " and "
       << c.R << endl;
}

// ============================================================================

// is a palidrome?

bool isPalidrome(string s) {
  return equal(s.begin(), next(s.begin(), s.size() / 2), s.rbegin());
}

// ============================================================================

// Enumerate all possible combinations

// Trick: initialize a vector of k 1's, then resize to 'n' and fill with 0's
// use next_permutation (in reverse) to "bitmask" and enumerate all options
int enumerate_combinations(int n, int k, vector<int> &A) {
  vector<int>  b(k, 1);
  b.resize(n, 0);

  // very basic, store count
  int cnt = 0;

  do {
    // pass over all n items, take/save/print the ones that match the "bitmask"
    for (int i = 0; i < n; i++) {
      if (b[i]) {
        cout << A[i] << " ";
      }
    }
    cnt++;
    cout << endl;

  } while (next_permutation(b.rbegin(), b.rend()));

  return cnt;
}

// ============================================================================

// Floyd Warshall, APSP, O(V^3)

// Floyd-warshall's APSP, initially g[i][j] is weight of path from i -> j
// for direct connections given; otherwise INF (0x3f3f3f3f ~ 1 bil).
// 'g' is represented as an adjacency matrix.  This algorithm is generally
```

```
// useable as long as number of vertices, V <= 400 (approx.)
REP(k, 101) {
   REP(i, 101) {
      REP(j, 101) { g[i][j] = min(g[i][j], g[i][k] + g[k][j]); }
   }
}

// =============================================================================

// C++ custom sort for pair (e.g. shown: desc second item, asc first item)

// Given pairs of the form pair<int, int>, wish to sort by descending second
// item, ascending first item
sort(ALL(ans), [](const ii &x, const ii &y) {
      return (x.second > y.second ||
              (x.second == y.second && x.first < y.first));
   });

// =============================================================================
```