```cpp
// 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;
```

```
67
68        M.assign(m, vector<int>(n));
69        for (int j = 0; j < n; j++) M[0][j] = j;
70
71        for (int i = 1; (1 << i) <= n; i++) {
72       for (int j = 0; (j + (1 << i)) <= n; j++) {
73          M[i][j] = (A[M[i - 1][j]] <= A[M[i - 1][j + (1 << (i - 1))]])
74             ? M[i - 1][j]
75             : M[i - 1][j + (1 << (i - 1))];
76       }
77        }
78     }
79
80     int query(int L, int R) {
81        int k = 31 - __builtin_clz(R - L + 1);
82        return (A[M[k][L]] <= A[M[k][R - (1 << k) + 1]])
83       ? M[k][L]
84       : M[k][R - (1 << k) + 1];
85     }
86
87  };
88
89  // ================================================================================
90
91  // LIS/LDS/DP O(n^2)
92
93  // DP N^2 to find and store length of longest ascending subsequence
94  for (int i = n - 1; i >= 0; i--) {
95     asc[i] = 1;
96     for (int j = i + 1; j < n; j++) {
97        // less than, for ascending subseq
98        if (A[i] < A[j]) {
99       asc[i] = max(asc[i], asc[j] + 1);
100       }
101    }
102 }
103
104 for (int i = n - 1; i >= 0; i--) {
105    desc[i] = 1;
106    for (int j = i + 1; j < n; j++) {
107       // greater than, for descending subseq
108       if (A[i] > A[j]) {
109      desc[i] = max(desc[i], desc[j] + 1);
110       }
111    }
112 }
113
114 // ================================================================================
115
116 // Longest ascending subsequence O(n log n)
117
118 vi LIS(vi &A) {
119   int n = (int)A.size();
120
121   // Uncomment to find for descending subsequence
122   // reverse(A.begin(), A.end());
123
124   vi last(n + 1), pos(n + 1), pred(n);
125   if (n == 0)
126     return vi();
127
128   int len = 1;
129   last[1] = A[pos[1] = 0];
130
131   for (int i = 1; i < n; i++) {
132     // 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;
  }
}
```

```
331
332
333 // ================================================================================
334
335 // Edmonds Karp - Max flow, O(V^3 * E) using adj mat
336
337
338 #define INF 0x3f3f3f3f
339
340 // mf: max flow (our solution, eventually)
341 // f : min f at the time
342 // s : source
343 // t : sink
344 int mf, f, s, t;
345
346 // parent, for BFS
347 vi p;
348
349 // max # of vertices we could have
350 #define MV 1000
351 int res[MV][MV];
352
353 void augment(int v, int lo) {
354   if (v == s) {
355     // base case: we reach source, and resolve 'f' to the minimum edge
356     f = lo;
357     return;
358   } else if (p[v] != -1) {
359     // wonder... is this check redundant? ==> no. It is needed because not
360     // necessarily have v -> u (back) for all u -> v
361
362     // recursive call
363     augment(p[v], min(lo, res[p[v]][v]));
364
365     // NOTE: changing with 'f', not 'lo'... why?
366     // because we're doing recursive calls, updating 'f' once we arrive at
367     // 's', at which point the global 'f' is updated, 'augment' returns up the
368     // stack, and these values need to be updated with that 'f', not just what
369     // 'lo' was at the time
370     //
371     // update forward res edges u -> v (i.e. p[v] -> v) with -f
372     // update backwrd res edges v -> u (i.e. v -> p[v]) with +f
373     res[p[v]][v] -= f;
374     res[v][p[v]] += f;
375   }
376 }
377
378 int main() {
379   ios_base::sync_with_stdio(false);
380   cin.tie(NULL);
381
382   // for each case, setup res, s, t
383   cin >> s >> t;
384
385   // init to 0, need to do this cause using adj matrix
386   REP(i, MV)
387   REP(j, MV) res[i][j] = 0;
388
389   // of the form:
390   // node_u node_v uv_capacity
391   int u, v, c;
392   while (cin >> u >> v >> c) {
393     cin >> u >> v >> c;
394     res[u][v] = c;
395   }
396
```

```
397   // max flow, initially 0
398   mf = 0;
399   for (;;) {
400     // don't forget this.  'f' needs to be reset each time
401     // or does it?... don't see why
402     // ==> YES: it is needed,
403     // ==> See below: <f needs to be 0>
404     f = 0;
405
406     // here, dist is the 'hop-count' from s to t
407     // but the actual 'weights' (i.e. capacities) of the edges are
408     // handled in 'res', initially equal to original values, but
409     // change over the course of the algorithm
410     vi dist(MV, INF);
411     dist[s] = 0;
412
413     // don't forget this part
414     p.assign(MV, -1);
415
416     queue<int> q;
417     q.push(s);
418     while (!q.empty()) {
419       auto u = q.front();
420       q.pop();
421       if (u == t) break;  // Exit condition: reach the sink
422
423       REP(v, MV) {
424         // Interpretation
425         // 1: res[u][v] > 0
426         // ==> v reachable from u
427         // 2: dist[v] == INF
428         // ==> unvisited
429         if (res[u][v] > 0 && dist[v] == INF)
430           dist[v] = dist[u] + 1, q.push(u), p[v] = u;
431       }
432     }
433     // BFS path has been created and stored by 'p', now augment.
434     // When this completes, we will know the minimum flow 'f' in this path ***IF
435     // ANY*** why IF ANY?  ==> because when 'f' == 0, we cannot send any more
436     // flow, so we are DONE
437     // ==> See above: <f needs to be 0>
438     augment(t, INF);
439     if (f == 0) break;
440
441     // we increase 'mf' maxflow solution for this case, by 'f' -> the flow that
442     // we still have the capacity to send.
443     mf += f;
444   }
445
446   // print 'mf' the maximum flow.
447   cout << mf << endl;
448
449   return 0;
450 }
451
452
453 // ================================================================================
454
455 // Find tree diameter
456
457 // graph representation of our tree, as an adjacency list
458 vvi g;
459
460 // Returns a pair<int, int>, first is the greatest depth found
461 // from 's', second is the node ID where that depth was found.
462 ii dfs(int s, int e) {
```

```
463   ii b = ii(0, s);
464   for (auto &u : g[s]) {
465
466     // only recurse on children (i.e. below 'u')
467     if (u != e) {
468       ii x = dfs(u, s);
469       x.L++;
470
471       // update, if greatest depth seen so far
472       if (x.L > b.L)
473         b = x;
474     }
475   }
476   return b;
477 }
478
479 void tree_diameter() {
480   ii b = dfs(0, 0);
481   ii c = dfs(b.R, b.R);
482   cout << "diameter is " << c.L << ", found between nodes " << b.R << " and "
483        << c.R << endl;
484 }
485
486 // ===========================================================================
487
488 // is a palidrome?
489
490 bool isPalidrome(string s) {
491   return equal(s.begin(), next(s.begin(), s.size() / 2), s.rbegin());
492 }
493
494 // ===========================================================================
495
496 // Enumerate all possible combinations
497
498 // Trick: initialize a vector of k 1's, then resize to 'n' and fill with 0's
499 // use next_permutation (in reverse) to "bitmask" and enumerate all options
500 int enumerate_combinations(int n, int k, vector<int> &A) {
501   vector<int>  b(k, 1);
502   b.resize(n, 0);
503
504   // very basic, store count
505   int cnt = 0;
506
507   do {
508     // pass over all n items, take/save/print the ones that match the "bitmask"
509     for (int i = 0; i < n; i++) {
510       if (b[i]) {
511         cout << A[i] << " ";
512       }
513     }
514     cnt++;
515     cout << endl;
516
517   } while (next_permutation(b.rbegin(), b.rend()));
518
519   return cnt;
520 }
521
522 // ===========================================================================
523
524 // Floyd Warshall, APSP, O(V^3)
525
526 // Floyd-warshall's APSP, initially g[i][j] is weight of path from i -> j
527 // for direct connections given; otherwise INF (0x3f3f3f3f ~ 1 bil).
528 // '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));
   });

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

```