

```

1 // Dijkstra
2
3 #include <bits/stdc++.h>
4 #define FR(i, n) for (int i = 0; i < (n); ++i)
5 using namespace std;
6
7 typedef vector<int> vi;
8 typedef pair<int, int> ii;
9
10 vector< vector<ii> > g;
11 vi D;
12 int n;
13
14 void dijkstra(int s) {
15     D.assign(n, -1);
16
17     priority_queue<ii, vector<ii>, greater<ii> > pq;
18     pq.emplace(0, s);
19
20     while (!pq.empty()) {
21         int u = pq.top().second, d = pq.top().first; pq.pop();
22
23         if (D[u] != -1) continue;
24         D[u] = d;
25
26         for (auto &i : g[u]) {
27             int v = i.second, w = i.first;
28             pq.emplace(d + w, v);
29         }
30     }
31 }
32
33 int main() {
34
35     cin >> n;
36     g.assign(n, vector<ii>());
37
38     // for each edge
39     int m = 5;
40     FR(i, m) {
41         int u, v, w;
42         cin >> u >> v >> w;
43         g[u].push_back(ii(w, v));
44         g[v].push_back(ii(w, u));
45     }
46
47     // run dijkstra from some src
48     dijkstra(0);
49
50     // path? for every edge from u -> v, if D[v] == D[u] + weight(u, v),
51     // then u -> v is an edge in "some" optimal path.
52 }
53
54
55
56 // =====
57
58 class RMQ {
59 public:
60
61     vector<int> A;
62     vector< vector<int> > M;
63     RMQ(const vector<int> &B) {
64         A = B;
65         int n = A.size();
66         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

```

```

133     int j = upper_bound(last.begin() + 1, last.begin() + len + 1, A[i]) -
134         last.begin();
135     pred[i] = (j - 1 > 0) ? pos[j - 1] : -1;
136     last[j] = A[pos[j] = i];
137     len = max(len, j);
138 }
139
140 int start = pos[len];
141 vi S(len);
142 for (int i = len - 1; i >= 0; i--) {
143     S[i] = A[start];
144     start = pred[start];
145 }
146
147 return S;
148 }
149
150 vi LIS_strict(vi &A) {
151     int n = (int)A.size();
152
153     // Uncomment to find for descending subsequence
154     // reverse(A.begin(), A.end());
155
156     vi last(n + 1), pos(n + 1), pred(n);
157     if (n == 0)
158         return vi();
159
160     int len = 1;
161     last[1] = A[pos[1] = 0];
162
163     for (int i = 1; i < n; i++) {
164         // lower_bound for strictly ascending
165         int j = lower_bound(last.begin() + 1, last.begin() + len + 1, A[i]) -
166             last.begin();
167         pred[i] = (j - 1 > 0) ? pos[j - 1] : -1;
168         last[j] = A[pos[j] = i];
169         len = max(len, j);
170     }
171
172     int start = pos[len];
173     vi S(len);
174     for (int i = len - 1; i >= 0; i--) {
175         S[i] = A[start];
176         start = pred[start];
177     }
178
179     return S;
180 }
181
182 // =====
183
184 // Tarjan's SCC
185
186 #define UNVISITED -1
187
188 int depth, num_scc;
189 vi num, lo, stk, vis;
190 vvi g;
191
192 // Tarjan's algorithm to find SCC's of directed graph; O(V+E).
193 // --> commonly used for pre-processing to contract digraph to DAG
194 void scc(int u) {
195     lo[u] = num[u] = depth++; // lo[u] <= num[u]
196
197     // push 'u' onto stack, and track explored vertices with 'vis'
198     stk.push_back(u);

```

```

199     vis[u] = 1;
200     for (auto &v : g[u]) {
201         if (num[v] == UNVISITED)
202             scc(v); // this part is amortized O(V)
203
204         // Condition to update:
205         if (vis[v])
206             lo[u] = min(lo[u], lo[v]);
207     }
208
209     // if root, i.e. start of an SCC
210     // Since only visited vertices may update lo[u], and initially we set lo[u] =
211     // num[u], then if lo[u] == num[u], we know 'u' is the root of this SCC. To
212     // access the members of this SCC, pop from our "stack" (i.e. 'stk', as
213     // vector), up to (and including) root 'u'.
214     if (lo[u] == num[u]) {
215         cout << "SCC #" << ++num_scc << ":";
216         for (;;) {
217             int v = stk.back();
218             stk.pop_back();
219             vis[v] = 0;
220             cout << " \n"[u == v] << v;
221             if (u == v)
222                 break;
223         }
224     }
225 }
226
227 int main() {
228
229     // Number of vertices
230     int V;
231     V = 10;
232
233     // Build adjacency list
234     g.assign(V, vi());
235     // ...
236
237     depth = num_scc = 0;
238     num.assign(V, UNVISITED);
239     lo.assign(V, 0);
240     vis.assign(V, 0);
241     REP(i, V) {
242         if (num[i] == UNVISITED) {
243             scc(i);
244         }
245     }
246
247     return 0;
248 }
249
250
251 // =====
252
253 // Find articulation points and bridges
254
255
256 // doesn't matter the value, just different
257 #define UNVISITED -1
258
259 // need these for APB algorithm, initialized in main()
260 int depth, root, children;
261 vi lo, num, parent;
262
263 // true for index i if i identifies an articulation point
264 vector<bool> art_points;

```

```

265
266 // adj list representation of graph
267 vector<vi> adj_list;
268
269 // Find Articulation Points and Bridges
270 void APB(int u) {
271     lo[u] = num[u] = depth++; // lo[u] <= num[u]
272     for (auto &v : adj_list[u]) {
273
274         if (num[v] == UNVISITED) {
275
276             // tree edge
277             parent[v] = u;
278             if (u == root)
279                 children++;
280
281             // Recurse
282             APB(v);
283
284             // articulation point
285             if (lo[v] >= num[u])
286                 art_points[u] = true;
287
288             // bridge
289             if (lo[v] > num[u])
290                 cout << "Edge " << u << " -> " << v << " is a bridge\n";
291
292             // update lo
293             lo[u] = min(lo[u], lo[v]);
294
295         } else if (v != parent[u]) {
296             // back edge, not a direct cycle
297             lo[u] = min(lo[u], num[v]);
298         }
299     }
300 }
301
302 void SolveForArticulationPointsAndBridges() {
303     // setup adjlist
304     int V;
305     V = 10;
306
307     // find articulation points and bridges
308     depth = 0;
309     num.assign(V, UNVISITED);
310     lo.assign(V, 0);
311     parent.assign(V, -1);
312     art_points.assign(V, false);
313
314     // first find bridges
315     cout << "Bridges:\n";
316     FOR(i, V) {
317         if (num[i] == UNVISITED) {
318             root = i, children = 0;
319             APB(i);
320             art_points[root] = (children > 1);
321         }
322     }
323
324     // points
325     cout << "Points:\n";
326     FOR(i, V) {
327         if (art_points[i])
328             cout << "Vertex " << i << endl;
329     }
330 }

```

```

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 backward 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

```



```

529 // useable as long as number of vertices, V <= 400 (approx.)
530 REP(k, 101) {
531     REP(i, 101) {
532         REP(j, 101) { g[i][j] = min(g[i][j], g[i][k] + g[k][j]); }
533     }
534 }
535
536 // =====
537
538 // C++ custom sort for pair (e.g. shown: desc second item, asc first item)
539
540 // Given pairs of the form pair<int, int>, wish to sort by descending second
541 // item, ascending first item
542 sort(ALL(ans), [](const ii &x, const ii &y) {
543     return (x.second > y.second ||
544            (x.second == y.second && x.first < y.first));
545 });
546
547 // =====
548

```