```
  dfn[i] = low[i] = ind++; stk[stks++] = i;
  for (int x = e.begin[i]; ~x; x = e.next[x]) {
   int j = e.dest[x]; if (!~dfn[j]) {
    dfs (e, j, i);
    if (low[i] > low[j]) low[i] = low[j];
   } else if (j != fa && low[i] > dfn[j]) low[i] = dfn
       [j]; }
  if (dfn[i] <= low[i]) {
   for (int j = -1; j != i; j = stk[--stks], comp[j] =
        size);
   ++size; } }
 void solve (const edge_list <MAXN, MAXM> &e, int n) {
  size = ind = stks = 0;
  std::fill (dfn, dfn + n, -1);
  for (int i = 0; i < n; ++i) if (!~dfn[i])
   dfs (e, i, -1); } };
```

4.4 Flow

4.4.1 Maximum flow

ISAP is better for sparse graphs, while Dinic is better for dense graphs.

```
template <int MAXN = 1000, int MAXM = 100000>
struct isap {
 struct flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], flow[
      MAXM];
  void clear (int n) { size = 0; std::fill (begin,
      begin + n, -1); }
  flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int f) {
   dest[size] = v; next[size] = begin[u]; flow[size] =
       f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; flow[size] =
       0; begin[v] = size++; } };
 int pre[MAXN], d[MAXN], gap[MAXN], cur[MAXN], que[
     MAXN], vis[MAXN];
 int solve (flow_edge_list &e, int n, int s, int t) {
  for (int i = 0; i < n; ++i) { pre[i] = d[i] = gap[i]
      = vis[i] = 0; cur[i] = e.begin[i]; }
  int l = 0, r = 0; que[0] = t; gap[0] = 1; vis[t] =
      true;
  while (l <= r) { int u = que[l++];
   for (int i = e.begin[u]; ~i; i = e.next[i])
    if (e.flow[i] == 0 && !vis[e.dest[i]]) {
     que[++r] = e.dest[i];
     vis[e.dest[i]] = true;
     d[e.dest[i]] = d[u] + 1;
     ++gap[d[e.dest[i]]]; } }
  for (int i = 0; i < n; ++i) if (!vis[i]) d[i] = n,
      ++gap[n];
  int u = pre[s] = s, v, maxflow = 0;
  while (d[s] < n) {
   v = n; for (int i = cur[u]; ~i; i = e.next[i])
    if (e.flow[i] && d[u] == d[e.dest[i]] + 1) {
     v = e.dest[i]; cur[u] = i; break; }
   if (v < n) {
    pre[v] = u; u = v;
    if (v == t) {
     int dflow = INF, p = t; u = s;
     while (p != s) { p = pre[p]; dflow = std::min (
         dflow, e.flow[cur[p]]); }
     maxflow += dflow; p = t;
     while (p != s) { p = pre[p]; e.flow[cur[p]] -=
         dflow; e.flow[cur[p] ^ 1] += dflow; } }
   } else {
    int mindist = n + 1;
    for (int i = e.begin[u]; ~i; i = e.next[i])
     if (e.flow[i] && mindist > d[e.dest[i]]) {
      mindist = d[e.dest[i]]; cur[u] = i; }
    if (!--gap[d[u]]) return maxflow;
    gap[d[u] = mindist + 1]++; u = pre[u]; } }
  return maxflow; } };
template <int MAXN = 1000, int MAXM = 100000>
struct dinic {
 struct flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], flow[
      MAXM];
  void clear (int n) { size = 0; std::fill (begin,
      begin + n, -1); }
  flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int f) {
   dest[size] = v; next[size] = begin[u]; flow[size] =
       f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; flow[size] =
       0; begin[v] = size++; } };
 int n, s, t, d[MAXN], w[MAXN], q[MAXN];
 int bfs (flow_edge_list &e) {
  std::fill (d, d + n, -1);
  int l, r; q[l = r = 0] = s, d[s] = 0;
  for (; l <= r; ++l)
   for (int k = e.begin[q[l]]; ~k; k = e.next[k])
    if (!~d[e.dest[k]] && e.flow[k] > 0) d[e.dest[k]]
        = d[q[l]] + 1, q[++r] = e.dest[k];
  return ~d[t] ? 1 : 0; }
 int dfs (flow_edge_list &e, int u, int ext) {
  if (u == t) return ext; int k = w[u], ret = 0;
  for (; ~k; k = e.next[k], w[u] = k) {
   if (ext == 0) break;
   if (d[e.dest[k]] == d[u] + 1 && e.flow[k] > 0) {
    int flow = dfs (e, e.dest[k], std::min (e.flow[k],
        ext));
    if (flow > 0) {
     e.flow[k] -= flow, e.flow[k ^ 1] += flow;
     ret += flow, ext -= flow; } } }
  if (!~k) d[u] = -1; return ret; }
 int solve (flow_edge_list &e, int n_, int s_, int t_)
     {
  int ans = 0; n = n_; s = s_; t = t_;
  while (bfs (e)) {
   for (int i = 0; i < n; ++i) w[i] = e.begin[i];
   ans += dfs (e, s, INF); }
  return ans; } };
```

4.4.2 Minimum cost flow

EK is better for sparse graphs, while ZKW is better for dense graphs.

```
template <int MAXN = 1000, int MAXM = 100000>
struct minimum_cost_flow {
 struct cost_flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], cost[
      MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin,
      begin + n, -1); }
  cost_flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int c, int f) {
   dest[size] = v; next[size] = begin[u]; cost[size] =
       c; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; cost[size] =
       -c; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, prev[MAXN], dist[MAXN], occur[MAXN];
 bool augment (cost_flow_edge_list &e) {
  std::vector <int> queue;
  std::fill (dist, dist + n, INF); std::fill (occur,
      occur + n, 0);
  dist[s] = 0; occur[s] = true; queue.push_back (s);
  for (int head = 0; head < (int)queue.size(); ++head)
      {
   int x = queue[head];
   for (int i = e.begin[x]; ~i; i = e.next[i]) {
    int y = e.dest[i];
    if (e.flow[i] && dist[y] > dist[x] + e.cost[i]) {
     dist[y] = dist[x] + e.cost[i]; prev[y] = i;
     if (!occur[y]) {
      occur[y] = true; queue.push_back (y); } } }
   occur[x] = false; }
  return dist[t] < INF; }
 std::pair <int, int> solve (cost_flow_edge_list &e,
     int n_, int s_, int t_) {
  n = n_; s = s_; t = t_; std::pair <int, int> ans =
      std::make_pair (0, 0);
  while (augment (e)) {
   int num = INF;
   for (int i = t; i != s; i = e.dest[prev[i] ^ 1])
    num = std::min (num, e.flow[prev[i]]);
   ans.first += num;
   for (int i = t; i != s; i = e.dest[prev[i] ^ 1]) {
    e.flow[prev[i]] -= num; e.flow[prev[i] ^ 1] += num
        ;
    ans.second += num * e.cost[prev[i]]; } }
  return ans; } };
template <int MAXN = 1000, int MAXM = 100000>
struct zkw_flow {
 struct cost_flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], cost[
      MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin,
      begin + n, -1); }
  cost_flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int c, int f) {
   dest[size] = v; next[size] = begin[u]; cost[size] =
       c; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; cost[size] =
       -c; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, tf, tc, dis[MAXN], slack[MAXN], visit[
     MAXN];
 int modlable() {
  int delta = INF;
  for (int i = 0; i < n; i++) {
   if (!visit[i] && slack[i] < delta) delta = slack[i
       ];
   slack[i] = INF; }
  if (delta == INF) return 1;
  for (int i = 0; i < n; i++) if (visit[i]) dis[i] +=
      delta;
  return 0; }
 int dfs (cost_flow_edge_list &e, int x, int flow) {
  if (x == t) { tf += flow; tc += flow * (dis[s] - dis
      [t]); return flow; }
  visit[x] = 1; int left = flow;
  for (int i = e.begin[x]; ~i; i = e.next[i])
   if (e.flow[i] > 0 && !visit[e.dest[i]]) {
    int y = e.dest[i];
    if (dis[y] + e.cost[i] == dis[x]) {
     int delta = dfs (e, y, std::min (left, e.flow[i])
         );
     e.flow[i] -= delta; e.flow[i ^ 1] += delta; left
         -= delta;
     if (!left) { visit[x] = false; return flow; }
    } else
     slack[y] = std::min (slack[y], dis[y] + e.cost[i]
         - dis[x]); }
  return flow - left; }
 std::pair <int, int> solve (cost_flow_edge_list &e,
     int n_, int s_, int t_) {
  n = n_; s = s_; t = t_; tf = tc = 0;
```

```
  std::fill (dis + 1, dis + n + 1, 0);
  do { do {
   std::fill (visit + 1, visit + n + 1, 0);
  } while (dfs (e, s, INF)); } while (!modlable ());
  return std::make_pair (tf, tc);
 } };
```

4.5 Matching

Tutte-Berge formula The theorem states that the size of a maximum matching of a graph $G = (V, E)$ equals

$$\frac{1}{2} \min_{U \subseteq V} (|U| - \text{odd}(G - U) + |V|) ,$$

where $\text{odd}(H)$ counts how many of the connected components of the graph H have an odd number of vertices.

Tutte theorem A graph, $G = (V, E)$, has a perfect matching if and only if for every subset U of V, the subgraph induced by $V - U$ has at most $|U|$ connected components with an odd number of vertices.

Hall's marriage theorem A family S of finite sets has a transversal if and only if S satisfies the marriage condition.

4.5.1 Blossom algorithm

Maximum matching for general graphs.

```
template <int MAXN = 500, int MAXM = 250000>
struct blossom {
 int match[MAXN], d[MAXN], fa[MAXN], c1[MAXN], c2[MAXN
     ], v[MAXN], q[MAXN];
 int *qhead, *qtail;
 struct {
  int fa[MAXN];
  void init (int n) { for(int i = 0; i < n; i++) fa[i]
       = i; }
  int find (int x) { if (fa[x] != x) fa[x] = find (fa[
      x]); return fa[x]; }
  void merge (int x, int y) { x = find (x); y = find (
      y); fa[x] = y; } } ufs;
 void solve (int x, int y) {
  if (x == y) return;
  if (d[y] == 0) {
   solve (x, fa[fa[y]]); match[fa[y]] = fa[fa[y]];
   match[fa[fa[y]]] = fa[y];
  } else if (d[y] == 1) {
   solve (match[y], c1[y]); solve (x, c2[y]);
   match[c1[y]] = c2[y]; match[c2[y]] = c1[y]; } }
 int lca (int x, int y, int root) {
  x = ufs.find (x); y = ufs.find (y);
  while (x != y && v[x] != 1 && v[y] != 0) {
   v[x] = 0; v[y] = 1;
   if (x != root) x = ufs.find (fa[x]);
   if (y != root) y = ufs.find (fa[y]); }
  if (v[y] == 0) std::swap (x, y);
  for (int i = x; i != y; i = ufs.find (fa[i])) v[i] =
       -1;
  v[y] = -1; return x; }
 void contract (int x, int y, int b) {
  for (int i = ufs.find (x); i != b; i = ufs.find (fa[
      i])) {
   ufs.merge (i, b);
   if (d[i] == 1) { c1[i] = x; c2[i] = y; *qtail++ = i
       ; } } }
 bool bfs (int root, int n, const edge_list <MAXN,
     MAXM> &e) {
  ufs.init (n); std::fill (d, d + MAXN, -1); std::fill
       (v, v + MAXN, -1);
  qhead = qtail = q; d[root] = 0; *qtail++ = root;
  while (qhead < qtail) {
   for (int loc = *qhead++, i = e.begin[loc]; ~i; i =
       e.next[i]) {
    int dest = e.dest[i];
    if (match[dest] == -2 || ufs.find (loc) == ufs.
        find (dest)) continue;
    if (d[dest] == -1)
     if (match[dest] == -1) {
      solve (root, loc); match[loc] = dest;
      match[dest] = loc; return 1;
     } else {
      fa[dest] = loc; fa[match[dest]] = dest;
      d[dest] = 1; d[match[dest]] = 0;
      *qtail++ = match[dest];
     } else if (d[ufs.find (dest)] == 0) {
      int b = lca (loc, dest, root);
      contract (loc, dest, b); contract (dest, loc, b)
          ; } } }
  return 0; }
 int solve (int n, const edge_list <MAXN, MAXM> &e) {
  std::fill (fa, fa + n, 0); std::fill (c1, c1 + n, 0)
      ;
  std::fill (c2, c2 + n, 0); std::fill (match, match +
       n, -1);
  int re = 0; for (int i = 0; i < n; i++)
   if (match[i] == -1) if (bfs (i, n, e)) ++re; else
       match[i] = -2;
  return re; } };
```

4.5.2 Blossom algorithm (weighted)

Maximum matching for general weighted graphs in $O(n^3)$ (1-based).
Usage:

1. Set n to the size of the vertices.
2. Execute `init`.
3. Set `g[][].w` to the weight of the edges.
4. Execute `solve`.
5. The first result is the answer, the second one is the number of matching pairs. Obtain the exact matching with `match[]`.

```
struct weighted_blossom {
 static const int INF = INT_MAX, MAXN = 400;
 struct edge{ int u, v, w; edge (int u = 0, int v = 0,
      int w = 0): u(u), v(v), w(w) {} };
 int n, n_x;
 edge g[MAXN * 2 + 1][MAXN * 2 + 1];
 int lab[MAXN * 2 + 1], match[MAXN * 2 + 1], slack[
     MAXN * 2 + 1], st[MAXN * 2 + 1], pa[MAXN * 2 +
     1];
 int flower_from[MAXN * 2 + 1][MAXN + 1], S[MAXN * 2 +
      1], vis[MAXN * 2 + 1];
 std::vector <int> flower[MAXN * 2 + 1]; std::queue <
     int> q;
 int e_delta (const edge &e) { return lab[e.u] + lab[e
     .v] - g[e.u][e.v].w * 2; }
 void update_slack (int u, int x) { if (!slack[x] ||
     e_delta (g[u][x]) < e_delta (g[slack[x]][x]))
     slack[x] = u; }
 void set_slack (int x) { slack[x] = 0; for (int u =
     1; u <= n; ++u) if(g[u][x].w > 0 && st[u] != x &&
      S[st[u]] == 0)
  update_slack(u, x); }
 void q_push (int x) {
  if (x <= n) q.push (x);
  else for (size_t i = 0; i < flower[x].size (); i++)
       q_push (flower[x][i]); }
 void set_st (int x, int b) {
  st[x] = b; if (x > n) for (size_t i = 0; i < flower[
      x].size (); ++i) set_st (flower[x][i], b); }
 int get_pr (int b, int xr){
  int pr = std::find (flower[b].begin (), flower[b].
      end (), xr) - flower[b].begin ();
  if (pr % 2 == 1) { std::reverse (flower[b].begin ()
      + 1, flower[b].end ()); return (int) flower[b].
      size () - pr;
  } else return pr; }
 void set_match (int u, int v){
  match[u] = g[u][v].v; if (u > n) {
   edge e = g[u][v]; int xr = flower_from[u][e.u], pr
       = get_pr (u, xr);
   for (int i = 0; i < pr; ++i) set_match (flower[u][i
       ], flower[u][i ^ 1]);
   set_match (xr, v); std::rotate (flower[u].begin (),
       flower[u].begin () + pr, flower[u].end ()); }
       }
 void augment (int u, int v) {
  for (; ; ) {
   int xnv = st[match[u]]; set_match (u, v);
   if (!xnv) return; set_match (xnv, st[pa[xnv]]);
   u = st[pa[xnv]], v = xnv; } }
 int get_lca (int u, int v){
  static int t = 0;
  for (++t; u || v; std::swap (u, v)) {
   if (u == 0) continue; if (vis[u] == t) return u;
   vis[u] = t; u = st[match[u]]; if (u) u = st[pa[u]];
       }
  return 0; }
 void add_blossom (int u, int lca, int v) {
  int b = n + 1; while (b <= n_x && st[b]) ++b;
  if (b > n_x) ++n_x;
  lab[b] = 0, S[b] = 0;
  match[b] = match[lca]; flower[b].clear ();
  flower[b].push_back (lca);
  for (int x = u, y; x != lca; x = st[pa[y]]) {
   flower[b].push_back (x), flower[b].push_back (y =
       st[match[x]]), q_push (y); }
  std::reverse (flower[b].begin () + 1, flower[b].end
       ());
  for (int x = v, y; x != lca; x = st[pa[y]]) {
   flower[b].push_back (x), flower[b].push_back (y =
       st[match[x]]), q_push(y); }
  set_st (b, b);
  for (int x = 1; x <= n_x; ++x) g[b][x].w = g[x][b].w
       = 0;
  for (int x = 1; x <= n; ++x) flower_from[b][x] = 0;
  for (size_t i = 0; i < flower[b].size (); ++i){
   int xs = flower[b][i];
   for (int x = 1; x <= n_x; ++x) if (g[b][x].w == 0
       || e_delta(g[xs][x]) < e_delta(g[b][x]))
    g[b][x] = g[xs][x], g[x][b] = g[x][xs];
   for (int x = 1;x <= n; ++x) if(flower_from[xs][x])
       flower_from[b][x] = xs; }
  set_slack (b); }
 void expand_blossom (int b) {
  for (size_t i = 0; i < flower[b].size (); ++i)
      set_st (flower[b][i], flower[b][i]);
  int xr = flower_from[b][g[b][pa[b]].u], pr = get_pr(
      b, xr);
  for (int i = 0; i < pr; i += 2) {
   int xs = flower[b][i], xns = flower[b][i + 1];
   pa[xs] = g[xns][xs].u; S[xs] = 1, S[xns] = 0;
   slack[xs] = 0, set_slack(xns); q_push(xns); }
  S[xr] = 1, pa[xr] = pa[b];
  for (size_t i = pr + 1; i < flower[b].size (); ++i)
       {
   int xs = flower[b][i]; S[xs] = -1, set_slack(xs); }
  st[b] = 0; }
 bool on_found_edge (const edge &e) {
  int u = st[e.u], v = st[e.v];
  if (S[v] == -1) {
```

```
    pa[v] = e.u, S[v] = 1; int nu = st[match[v]];
    slack[v] = slack[nu] = 0; S[nu] = 0, q_push(nu);
   } else if(S[v] == 0) {
    int lca = get_lca(u, v);
    if (!lca) return augment(u, v), augment(v, u), true
        ;
    else add_blossom(u, lca, v); }
   return false; }
  bool matching () {
   memset (S + 1, -1, sizeof (int) * n_x);
   memset (slack + 1, 0, sizeof (int) * n_x);
   q = std::queue <int> ();
   for (int x = 1; x <= n_x; ++x) if (st[x] == x && !
       match[x]) pa[x] = 0, S[x] = 0, q_push (x);
   if (q.empty ()) return false;
   for (; ; ) {
    while (q.size ()) {
     int u = q.front (); q.pop ();
     if (S[st[u]] == 1) continue;
     for (int v = 1; v <= n; ++v) if (g[u][v].w > 0 &&
         st[u] != st[v]) {
      if(e_delta (g[u][v]) == 0){
       if (on_found_edge (g[u][v])) return true;
      } else update_slack (u, st[v]); } }
    int d = INF;
    for (int b = n + 1; b <= n_x; ++b) if(st[b] == b &&
         S[b] == 1) d = std::min (d, lab[b] / 2);
    for (int x = 1; x <= n_x; ++x) if(st[x] == x &&
        slack[x]) {
     if (S[x] == -1) d = std::min (d, e_delta (g[slack[
         x]][x]));
     else if (S[x] == 0) d = std::min (d, e_delta (g[
         slack[x]][x]) / 2); }
    for (int u = 1; u <= n; ++u) {
     if (S[st[u]] == 0){
      if(lab[u] <= d) return 0;
      lab[u] -= d;
     } else if (S[st[u]] == 1) lab[u] += d; }
    for (int b = n + 1; b <= n_x; ++b)
     if (st[b] == b) {
      if (S[st[b]] == 0) lab[b] += d * 2;
      else if (S[st[b]] == 1) lab[b] -= d * 2; }
    q = std::queue <int> ();
    for (int x = 1; x <= n_x; ++x)
     if (st[x] == x && slack[x] && st[slack[x]] != x &&
         e_delta(g[slack[x]][x]) == 0)
      if(on_found_edge (g[slack[x]][x]))return true;
    for (int b = n + 1; b <= n_x; ++b) if (st[b] == b
        && S[b] == 1 && lab[b] == 0) expand_blossom(b);
         }
   return false; }
  std::pair <long long, int> solve () {
   memset (match + 1, 0, sizeof (int) * n); n_x = n;
   int n_matches = 0; long long tot_weight = 0;
   for (int u = 0; u <= n; ++u) st[u] = u, flower[u].
       clear();
   int w_max = 0;
   for (int u = 1; u <= n; ++u) for (int v = 1; v <= n;
        ++v) {
    flower_from[u][v] = (u == v ? u : 0); w_max = std::
        max (w_max, g[u][v].w); }
   for (int u = 1; u <= n; ++u) lab[u] = w_max;
   while (matching ()) ++n_matches;
   for (int u = 1; u <= n; ++u) if (match[u] && match[u
       ] < u) tot_weight += g[u][match[u]].w;
   return std::make_pair (tot_weight, n_matches); }
  void init () { for (int u = 1; u <= n; ++u) for (int
      v = 1; v <= n; ++v) g[u][v] = edge (u, v, 0); }
      };
```

4.5.3 Hopcoft-Karp algorithm

Unweighted maximum matching for bipartite graphs in $O(m\sqrt{n})$.

```
template <int MAXN = 100000, int MAXM = 100000>
struct hopcoft_karp {
 int mx[MAXN], my[MAXM], lv[MAXN];
 bool dfs (edge_list <MAXN, MAXM> &e, int x) {
  for (int i = e.begin[x]; ~i; i = e.next[i]) {
   int y = e.dest[i], w = my[y];
   if (!~w || (lv[x] + 1 == lv[w] && dfs (e, w))) {
    mx[x] = y; my[y] = x; return true; } }
  lv[x] = -1; return false; }
 int solve (edge_list <MAXN, MAXM> &e, int n, int m) {
  std::fill (mx, mx + n, -1); std::fill (my, my + m,
      -1);
  for (int ans = 0; ; ) {
   std::vector <int> q;
   for (int i = 0; i < n; ++i)
    if (mx[i] == -1) {
     lv[i] = 0; q.push_back (i);
    } else lv[i] = -1;
   for (int head = 0; head < (int) q.size(); ++head) {
    int x = q[head];
    for (int i = e.begin[x]; ~i; i = e.next[i]) {
     int y = e.dest[i], w = my[y];
     if (~w && lv[w] < 0) { lv[w] = lv[x] + 1; q.
         push_back (w); } } }
   int d = 0; for (int i = 0; i < n; ++i) if (!~mx[i]
       && dfs (e, i)) ++d;
   if (d == 0) return ans; else ans += d; } } };
```

4.5.4 Kuhn-Munkres algorithm

Weighted maximum matching on bipartition graphs. Input n and w. Collect the matching in `m[]`. The graph is 1-based.

```
template <int MAXN = 500>
struct kuhn_munkres {
 int n, w[MAXN][MAXN], lx[MAXN], ly[MAXN], m[MAXN],
     way[MAXN], sl[MAXN];
 bool u[MAXN];
 void hungary(int x) {
  m[0] = x;  int j0 = 0;
  std::fill (sl, sl + n + 1, INF); std::fill (u, u + n
      + 1, false);
  do {
   u[j0] = true; int i0 = m[j0], d = INF, j1 = 0;
   for (int j = 1; j <= n; ++j)
    if (u[j] == false) {
     int cur = -w[i0][j] - lx[i0] - ly[j];
     if (cur < sl[j]) { sl[j] = cur; way[j] = j0; }
     if (sl[j] < d) { d = sl[j]; j1 = j; } }
   for (int j = 0; j <= n; ++j) {
    if (u[j]) { lx[m[j]] += d; ly[j] -= d; }
    else sl[j] -= d; }
   j0 = j1; } while (m[j0] != 0);
  do {
   int j1 = way[j0]; m[j0] = m[j1]; j0 = j1;
  } while (j0); }
 int solve() {
  for (int i = 1; i <= n; ++i) m[i] = lx[i] = ly[i] =
      way[i] = 0;
  for (int i = 1; i <= n; ++i) hungary (i);
  int sum = 0; for (int i = 1; i <= n; ++i) sum += w[m
      [i]][i];
  return sum; } };
```

4.6 Path

4.6.1 Lindström-Gessel-Viennot lemma

Let G be a locally finite directed acyclic graph. This means that each vertex has finite degree, and that G contains no directed cycles. Consider base vertices $A = \{a_1, \ldots, a_n\}$ and destination vertices $B = \{b_1, \ldots, b_n\}$, and also assign a weight ω_e to each directed edge e. These edge weights are assumed to belong to some commutative ring. For each directed path P between two vertices, let $\omega(P)$ be the product of the weights of the edges of the path. For any two vertices a and b, write $e(a, b)$ for the sum $e(a, b) = \sum_{P: a \to b} \omega(P)$ over all paths from a to b.

With this setup, write:

$$M = \begin{pmatrix} e(a_1, b_1) & e(a_1, b_2) & \cdots & e(a_1, b_n) \\ e(a_2, b_1) & e(a_2, b_2) & \cdots & e(a_2, b_n) \\ \vdots & \vdots & \ddots & \vdots \\ e(a_n, b_1) & e(a_n, b_2) & \cdots & e(a_n, b_n) \end{pmatrix}.$$

An n-tuple of non-intersecting paths from A to B means an n-tuple $(P_1, \ldots, P_n)$ of paths in G with the following properties:

1. There exists a permutation σ of $\{1, 2, ..., n\}$ such that, for every i, the path P_i is a path from a_i to $b_{\sigma(i)}$.
2. Whenever $i \neq j$, the paths P_i and P_j have no two vertices in common (not even endpoints).

Given such an n-tuple $(P_1, \ldots, P_n)$, we denote by $\sigma(P)$ the permutation of σ from the first condition.

The Lindström-Gessel-Viennot lemma then states that the determinant of M is the signed sum over all n-tuples $P = (P_1, \ldots, P_n)$ of non-intersecting paths from A to B:

$$\det(M) = \sum_{(P_1, \ldots, P_n):\ A \to B} \operatorname{sign}(\sigma(P)) \prod_{i=1}^{n} \omega(P_i).$$

That is, the determinant of M counts the weights of all n-tuples of non-intersecting paths starting at A and ending at B, each affected with the sign of the corresponding permutation of $(1, 2, \ldots, n)$, given by P_i taking a_i to $b_{\sigma(i)}$.

In particular, if the only permutation possible is the identity (i.e., every n-tuple of non-intersecting paths from A to B takes a_i to b_i for each i) and we take the weights to be 1, then $\det(M)$ is exactly the number of non-intersecting n-tuples of paths starting at A and ending at B.

4.7 Tree

4.7.1 Optimum branching

The index of the root is 1. Check `(sel[i], i)` for `i` in `[2..n]` for the result.

```
template <int MAXN = 1000>
struct optimum_branching {
 int from[MAXN][MAXN * 2], n, m, edge[MAXN][MAXN * 2];
 int sel[MAXN * 2], fa[MAXN * 2], vis[MAXN * 2];
 int getfa (int x) { if(x == fa[x]) return x; return
     fa[x] = getfa(fa[x]); }
 void liuzhu () {
  fa[1] = 1; for (int i = 2; i <= n; ++i) {
   sel[i] = 1; fa[i] = i;
   for (int j = 1; j <= n; ++j) if (fa[j] != i)
    if (from[j][i] = i, edge[sel[i]][i] > edge[j][i])
     sel[i] = j; }
  int limit = n, prelimit = 0; do {
   prelimit = limit; memset (vis, 0, sizeof(vis)); vis
       [1] = 1;
```

Problem I. Magic Potion

Input file: `standard input`
Output file: `standard output`

There are n heroes and m monsters living in an island. The monsters became very vicious these days, so the heroes decided to diminish the monsters in the island. However, the i-th hero can only kill one monster belonging to the set M_i. Joe, the strategist, has k bottles of magic potion, each of which can buff one hero's power and let him be able to kill one more monster. Since the potion is very powerful, a hero can only take at most one bottle of potion.

Please help Joe find out the maximum number of monsters that can be killed by the heroes if he uses the optimal strategy.

Input

The first line contains three integers n, m, k ($1 \le n, m, k \le 500$) — the number of heroes, the number of monsters and the number of bottles of potion.

Each of the next n lines contains one integer t_i, the size of M_i, and the following t_i integers $M_{i,j}$ ($1 \le j \le t_i$), the indices (1-based) of monsters that can be killed by the i-th hero ($1 \le t_i \le m, 1 \le M_{i,j} \le m$).

Output

Print the maximum number of monsters that can be killed by the heroes.

Examples

standard input	standard output
3 5 2 4 1 2 3 5 2 2 5 2 1 2	4
5 10 2 2 3 10 5 1 3 4 6 10 5 3 4 6 8 9 3 1 9 10 5 1 3 6 7 10	7

Nowadays a lot of Kejin games (the games which are free to get and play, but some items or characters are unavailable unless you pay for it) appeared. For example, Love Live, Kankore, Puzzle & Dragon, Touken Ranbu and Kakusansei Million Arthur (names are not listed in particular order) are very typical among them. Their unbelievably tremendous popularity has become a hot topic, and makes considerable profit every day. You are now playing another Kejin game. In this game, your character has a skill graph which decides how can you gain skills. Particularly speaking, skill graph is an oriented graph, vertices represent skills, and arcs show their relationship — if an arc from A to B exists in the graph (i.e. B has a dependency on A), you need to get skill A before you are ready to gain skill B. If a skill S has more than one dependencies, they all need to be got firstly in order to gain S. Note that there is no cycles in the skill graph, and no two same arcs. Getting a skill takes time and energy, especially for those advanced skills appear very deep in the skill graph. However, as an RMB player, you know that in the game world money could distort even basic principles. For each arc in skill graph, you can "Ke" (which means to pay) some money to erase it. Further, for each skill, you could even "Ke" a sum of money to gain it directly in defiance of any dependencies! As you have neither so much leisure time to get skills nor sufficient money, you decide to balance them. All costs, including time, energy or money, can be counted in the unit "TA". You calculate costs for all moves (gaining a skill in normal way, erasing an arc and gaining a skill directly). Note that all costs are non-negative integers. Then, you want to know the minimum cost to gain a particular skill S if you haven't get any skills initially. Solve this problem to make your game life more joyful and . . . economical.

Input

The input consists of no more than 10 test cases, and it starts with a single integer indicating the number of them. The first line of each test case contains 3 positive integers N ($1 \le N \le 500$), M ($1 \le M \le 10000$) and S, representing the number of vertices and arcs in the skill graph, and the index of the skill you'd like to get. Vertices are indexed from 1 to N, each representing a skill. Then M lines follow, and each line consists of 3 integers A, B and C, indicating that there is an arc from skill A

to skill B, and C ($1 \le C \le 1000000$) TAs are needed to erase this arc. The next line contains N integers representing the cost to get N skills in normal way. That means, the i-th integer representing the cost to get the i-th skill after all its dependencies are handled. The last line also contains N integers representing the cost to get N skills directly by "Ke". These 2N integers are no more than 1000000.

Output

For each test case, output your answer, the minimum total cost to gain skill S, in a single line.

Sample Input

```
2
555
125
135
248
4 5 10
3 5 15
3 5 7 9 11
100 100 100 200 200
555
125
135
248
4 5 10
3 5 15
3 5 7 9 11
5 5 5 50 50
```

Sample Output

31

26

Sponsor

Coding Contest HDU - 5988

A coding contest will be held in this university, in a huge playground. The whole playground would be divided into N blocks, and there would be M directed paths linking these blocks. The i-th path goes from the u_i-th block to the v_i-th block. Your task is to solve the lunch issue. According to the arrangement, there are s_i competitors in the i-th block. Limited to the size of table, b_i bags of lunch including breads, sausages and milk would be put in the i-th block. As a result, some competitors need to move to another block to access lunch. However, the playground is temporary, as a result there would be so many wires on the path.
For the i-th path, the wires have been stabilized at first and the first competitor who walker through it would not break the wires. Since then, however, when a person go through the i - th path, there is a chance of p_i to touch
the wires and affect the whole networks. Moreover, to protect these wires, no more than c_i competitors are allowed to walk through the i-th path.
Now you need to find a way for all competitors to get their lunch, and minimize the possibility of network crashing.

Input

The first line of input contains an integer t which is the number of test cases. Then t test cases follow.
For each test case, the first line consists of two integers N ($N \le 100$) and M ($M \le 5000$).
Each of the next N lines contains two integers si and b_i ($s_i , b_i \le 200$).
Each of the next M lines contains three integers u_i , v_i and c_i($c_i \le 100$) and a float-point number p_i($0 < p_i < 1$).
It is guaranteed that there is at least one way to let every competitor has lunch.

Output

For each turn of each case, output the minimum possibility that the networks would break down. Round it to 2 digits.

Sample

Input	copy	Output	copy
1 4 4 2 0 0 3 3 0 0 3 1 2 5 0.5 3 2 5 0.5 1 4 5 0.5 3 4 5 0.5		0.50	