ACM-ICPC 팀노트

exytring

배지운 안건주 이영수

math utils

```
/* PRIME NUMBERS
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,
103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337,
347, 349, 353, 359, 367, 373, 379, 383, 389, 397,
401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523,
541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811,
821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953,
967, 971, 977, 983, 991, 997
density of prime numbers : x / log x  (lim x -> INF)

PI
3.14159265358979323846264338327950288419716939937510582097494459230781640

*/

// return a % b (positive value)
int mod(int a, int b) {
    return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
   int tmp;
   while(b){a%=b; tmp=a; a=b; b=tmp;}
   return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
   return a/gcd(a,b)*b;
}
```

```
// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
   int xx = y = 0;
   int yy = x = 1;
   while (b) {
      int q = a/b;
      int t = b; b = a%b; a = t;
      t = xx; xx = x-q*xx; x = t;
      t = yy; yy = y-q*yy; y = t;
   }
   return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
   int x, y;
   VI solutions;
   int d = extended_euclid(a, n, x, y);
   if (!(b%d)) {
      x = mod (x*(b/d), n);
      for (int i = 0; i < d; i++)
     solutions.push_back(mod(x + i*(n/d), n));
    }
    return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
   int x, y;
   int d = extended_euclid(a, n, x, y);
   if (d > 1) return -1;
   return mod(x,n);
}
```

dijkstra

```
typedef pair<int,int> PII;
vector<vector<PII> > edges;
vector<int> dist, dad;

// N: number of graph list
// s: start of graph
// t: tail of graph
void dijkstra(int N, int s, int t) {
   // use priority queue in which top element has the "smallest" priority
   priority_queue<PII, vector<PII>, greater<PII> > Q;
   dist.assign(N, INF);
   dad.assign(N, -1);

   Q.push(make_pair (0, s));
   dist[s] = 0;
   while (!Q.empty()) {
       PII p = Q.top();
       Q.pop();
       if (p.second == t) break;
       int here = p.second;
       for (vector<PII>::iterator it = edges[here].begin(); it != edges[here].end(); it++){
           if (dist[here] + it->first < dist[it->second]){
               dist[it->second] = dist[here] + it->first;
               dad[it->second] = here;
               Q.push (make_pair (dist[it->second], it->second));
           }
       }
   }
}

int main(){
   int V, E, start;
   scanf("%d %d %d", &V, &E, &start);
   edges.resize(V);
   for (int i = 0; i < E; i++) {
       int u, v, w; scanf("%d %d %d", &u, &v, &w);
       edges[--u].push_back( make_pair(w, --v) );
   }
   dijkstra(V, 0, -1);

   for (int i = 0; i < V; i++)
       printf("%d\n", dist[i]);
}
```

floyd warshall

```
void floydWarshall(vector< vector<type> >& map)
{
   for(int k = 0; k< map.size(); k++)
       for(int i = 0; i < map.size(); i++)
           for(int j = 0; j< map.size(); j++)
               if(map[i][j] > map[i][k] + map[k][j])
                   map[i][j] = map[i][k] + map[k][j];
}
```

longest incresing subsequence

```
typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;
#define STRICTLY_INCREASNG

VI lis(VI v) {
   VPII best;
   VI dad(v.size(), -1);

   for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
       PII item = make_pair(v[i], 0);
       VPII::iterator it = lower_bound(best.begin(), best.end(), item);
       item.second = i;
#else
       PII item = make_pair(v[i], i);
       VPII::iterator it = upper_bound(best.begin(), best.end(), item);
#endif
       if (it == best.end()) {
           dad[i] = (best.size() == 0 ? -1 : best.back().second);
           best.push_back(item);
       }else {
           dad[i] = dad[it->second];
           *it = item;
       }
   }

   VI ret;
   for (int i = best.back().second; i >= 0; i = dad[i])
       ret.push_back(v[i]);
   reverse(ret.begin(), ret.end());
   return ret;
}
```

segment tree (with lazy)

```
lld tree[MX * 4];
lld lazy[MX * 4];
int len, r;

void build(int N) {
   len = 1; r = 1;
   while (r <= N) {
       r *= 2;
       len += r;
   }
   int i;
   for (i = len - r + 1; i <= len - r + N; i++)
       scanf("%lld", tree + i);
   for (i = len - r + N + 1; i <= len; i++)
       tree[i] = 0;
   for (i = len - r; i >= 1; i--)
       tree[i] = tree[i * 2] + tree[i * 2 + 1];
   for (i = 1; i <= len; i++) lazy[i] = 0;
}

void update(int node, int start, int end, int l, int r, int val) {
   if (lazy[node] != 0) {
       tree[node] += (end - start + 1) * lazy[node];
       if (start != end) {
           lazy[node * 2] += lazy[node];
           lazy[node * 2 + 1] += lazy[node];
       }
       lazy[node] = 0;
   }
   if (start > r || end < l || start >end) return;
   if (l <= start && end <= r) {
       tree[node] += (end - start + 1) * val;
       if (start != end) {
           lazy[node * 2] += val;
           lazy[node * 2 + 1] += val;
       }
       return;
   }
   int mid = (start + end) / 2;
   update(node * 2, start, mid, l, r, val);
   update(node * 2 + 1, mid + 1, end, l, r, val);
   tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

```
lld query(int node, int start, int end, int l, int r) {
    if (start > end || start > r || end < l) return 0; //out of range

    if (lazy[node] != 0) {
        tree[node] += (end - start + 1) * lazy[node];
        if (start != end) {
            lazy[node * 2] += lazy[node];
            lazy[node * 2 + 1] += lazy[node];
        }
        lazy[node] = 0;
    }
    if (l <= start && end <= r)
        return tree[node];

    int mid = (start + end) / 2;
    return query(node * 2, start, mid, l, r) + query(node * 2 + 1, mid + 1, end, l, r);
}

int main(void) {
    int N, M, K; scanf("%d %d %d", &N, &M, &K);
    build(N);

    for (int i = 0; i<M + K; i++) {
        int type; scanf("%d", &type);
        if (type == 1) { //update
            int b, c, d; scanf("%d %d %d", &b, &c, &d);
            if (b > c) {
                int t = b; b = c; c = t;
            }
            update(1, 0, r- 1, --b, --c, d);
        }
        else { //query
            int b, c; scanf("%d %d", &b, &c);
            if (b > c) {
                int t = b; b = c; c = t;
            }
            printf("%lld\n", query(1, 0, r- 1, --b ,--c));
        }
    }
    return 0;
}
```

bitpartite matching

```
vector<vector<int> > adj;
vector<int> a_match, b_match;
vector<bool> visited;

bool dfs(int a){
   if (visited[a]) return false;
   visited[a] = true;

   for (auto b: adj[a]) {
       if (b_match[b] == -1 || dfs(b_match[b])) {
           a_match[a] = b;
           b_match[b] = a;
           return true;
       }
   }

   return false;
}
int bipartiteMatch(int n, int m){
   a_match.assign(n, -1);
   b_match.assign(m, -1);

   int ret = 0;
   for (int s = 0; s < n; ++s){
       visited.assign(n, false);
       if (dfs(s)) ++ret;
   }
   return ret;
}

int main() {
   int n, m; scanf("%d %d", &n, &m);
   adj.resize(n);

   for (int i=0; i<n; i++) {
       int k; scanf("%d", &k);

       for (int j=0; j<k; j++) {
           int x;  scanf("%d", &x);
           adj[i].push_back( x-1 );
       }
   }
   printf("%d", bipartiteMatch(n, m));
}
```

network flow (ford fulkerson)

```
// O(N^3)
#include <math.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <unordered_map>

#define INF 1987654321
using namespace std;
typedef int node_value;
struct Node; struct Edge;

struct Edge {
   Node* target; Edge* reverse;
   int capacity, flow;

   Edge() { this->flow = 0; }
   Edge(Node* target, int capacity) {
       this->target = target;
       this->capacity = capacity;
       this->flow = 0;
   }
   int getRestCapacity() { return capacity - flow; }
   void push(int amt){ flow += amt;  this->reverse->flow -= amt; }
};

struct Node {
   vector<Edge*> edges;
   node_value value;

   Node() {}
   Node(node_value val) { this->value = val; }

   void connectTo(Node* target, int capacity) {
       Edge *e = new Edge(target, capacity);
       Edge *e_rev = new Edge(this, 0);
       e->reverse = e_rev;
       e_rev->reverse = e;

       this->edges.push_back( e );
       target->edges.push_back( e_rev );
   }
};
```

```
int networkFlow(Node* source, Node* sink) {
    int totalAmount = 0;
    while (true) {
        queue< Node* > que;
        unordered_map<Node*, Edge*> track;
        que.push( source );
        while (!que.empty() && track[sink] == nullptr) {
            Node* here = que.front();
            que.pop();
            for (int k=0; k<here->edges.size(); k++) {
                Edge* e = here->edges[k];
                if (e->getRestCapacity() > 0 && track[e->target] == nullptr) {
                    que.push( e->target );
                    track[ e->target ] = e->reverse;
                }
            }
        }
        if (track[sink] == nullptr) break;

        int amount = INF;
        for (Node* n = sink; n != source; n = track[n]->target)
            amount = min(amount, track[n]->reverse->getRestCapacity());

        // 유량 갱신
        for (Node* n = sink; n != source; n = track[n]->target)
            track[n]->push( -amount );

        totalAmount += amount;
    }
    return totalAmount;
}

int main() {
    int N, M; scanf("%d %d", &N, &M);
    Node* source = new Node;
    Node* sink = new Node;
    vector<Node*> cows(N), sheds(M);

    for (int i=0; i<N; i++) {
        cows[i] = new Node(i+1);
        source->connectTo(cows[i], 1);
    }
    for (int i=0; i<M; i++) {
        sheds[i] = new Node(i+1);
        sheds[i]->connectTo(sink, 1);
    }
```

```
    for (int i=0; i<N; i++) {
        int si; scanf("%d", &si);
        for (int j=0; j<si; j++) {
            int x; scanf("%d", &x);
            cows[i]->connectTo(sheds[x-1], 1);
        }
    }

    int ret = networkFlow(source, sink);
    printf("%d", ret);
}
```

min cost max flow

```
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
// OUTPUT:
//     - (maximum flow value, minimum cost value)

#include <math.h>
#include <vector>
#include <iostream>
#include <limits>

using namespace std;
typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;
L min(L a, L b) { return a < b ? a : b; }
```

```
struct MinCostMaxFlow {
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;
    VPII dad;
    MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

    void AddEdge(int from, int to, L cap, L cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }

    void Relax(int s, int k, L cap, L cost, int dir) {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k]) {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }

    L Dijkstra(int s, int t) {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;

        while (s != -1) {
            int best = -1;
            found[s] = true;
            for (int k = 0; k < N; k++) {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }

        for (int k = 0; k < N; k++)
            pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }
```

```
    pair<L, L> GetMaxFlow(int s, int t) {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t)) {
            totflow += amt;
            for (int x = t; x != s; x = dad[x].first) {
                if (dad[x].second == 1) {
                    flow[dad[x].first][x] += amt;
                    totcost += amt * cost[dad[x].first][x];
                } else {
                    flow[x][dad[x].first] -= amt;
                    totcost -= amt * cost[x][dad[x].first];
                }
            }
        }
        return make_pair(totflow, totcost);
    }
};


int main() {
    int N, A, B;
    scanf("%d %d %d", &N, &A, &B);

    MinCostMaxFlow mcmf(N+4);
    // set source: 0, sink: 1, A:2, B:3

    for (int i = 0; i < N; i++) {
        int K, Da, Db;
        scanf("%d %d %d", &K, &Da, &Db);

        mcmf.AddEdge(0, i+4, K, 0);
        mcmf.AddEdge(i+4, 2, INF, Da);
        mcmf.AddEdge(i+4, 3, INF, Db);
    }

    mcmf.AddEdge(2, 1, A, 0);
    mcmf.AddEdge(3, 1, B, 0);

    pair<L, L> ret = mcmf.GetMaxFlow(0, 1);
    printf("%lld %lld", ret.first, ret.second);

}
```

convex hull

```
// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm.  Eliminate redundant points from the hull if REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
//   INPUT: a vector of input points, unordered.
//   OUTPUT:a vector of points in the convex hull, counterclockwise,
//            starting with bottommost/leftmost point

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;

#define REMOVE_REDUNDANT
typedef double T;
const T EPS = 1e-7;

struct PT {
   T x, y;
   PT() {}
   PT(T x, T y) : x(x), y(y) {}
   bool operator<(const PT &rhs) const { return make_pair(y,x) < make_pair(rhs.y,rhs.x); }
   bool operator==(const PT &rhs) const { return make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }


#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
   return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif


void ConvexHull(vector<PT> &pts) {
   sort(pts.begin(), pts.end());
   pts.erase(unique(pts.begin(), pts.end()), pts.end());

   vector<PT> up, dn;

   for (int i = 0; i < pts.size(); i++) {
       while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
       while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
       up.push_back(pts[i]);
       dn.push_back(pts[i]);
   }

   pts = dn;
   for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);
```

```
#ifdef REMOVE_REDUNDANT
   if (pts.size() <= 2) return;
   dn.clear();
   dn.push_back(pts[0]);
   dn.push_back(pts[1]);
   for (int i = 2; i < pts.size(); i++) {
       if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
       dn.push_back(pts[i]);
   }
   if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
       dn[0] = dn.back();
       dn.pop_back();
   }
   pts = dn;
#endif
}
```

suffix array

```
// Suffix array construction in O(L log^2 L) time.    Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:     string s
//
// OUTPUT:    array suffix[] such that suffix[i] = index (from 0 to L-1)
//                    of substring s[i...L-1] in the list of sorted suffixes.
//                    That is, if we take the inverse of the permutation suffix[],
//                    we get the actual suffix array.

#include <vector>
#include <iostream>
#include <string>
using namespace std;

struct SuffixArray {
   const int L;
   string s;
   vector<vector<int> > P;
   vector<pair<pair<int,int>,int> > M;

   SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {
      for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
      for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
         P.push_back(vector<int>(L, 0));
         for (int i = 0; i < L; i++)
             M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);

         sort(M.begin(), M.end());
         for (int i = 0; i < L; i++)
            P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;
      }
   }

   vector<int> GetSuffixArray() { return P.back(); }
```

```
    // returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
    int LongestCommonPrefix(int i, int j) {
        int len = 0;
        if (i == j) return L - i;
        for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
            if (P[k][i] == P[k][j]) {
                i += 1 << k;
                j += 1 << k;
                len += 1 << k;
            }
        }
        return len;
    }
};



int main() {
    // bobocel is the 0'th suffix
    //    obocel is the 5'th suffix
    //     bocel is the 1'st suffix
    //        ocel is the 6'th suffix
    //         cel is the 2'nd suffix
    //            el is the 3'rd suffix
    //             l is the 4'th suffix
    SuffixArray suffix("bobocel");
    vector<int> v = suffix.GetSuffixArray();

    // Expected output: 0 5 1 6 2 3 4
    //                                  2
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```