Competitive Programming Team Book

21 de maio de 2025

Conteúdo

1 dp

1 LIS.cpp

```
#include "../presets/base.cpp"
// Lis O(n log k)
void print_LIS(vector<int> &P, vector<int> &A, int i) {
  if (P[i] == -1) cout << A[i];
  print_LIS(P, A, P[i]);
  cout <<" "<<A[i];
}

int  n,k = 0; // n is the number of elements and k is the length
    length of actual LIS.
vector<int> L(n,0), L_id(n), p(n);
// L has the values of the last element of LIS with length i
// L_id is the position of the element L[i] in original array
// P is the position of the element that addded before this
// number.
// If we just need the length of the LIS, the vectors L_id and p
    are unnecessary
int lis(vector<int> &A){
  for (int i = 0; i < n; ++i) {          // O(n)
    int pos = lower_bound(L.begin(), L.begin()+k, A[i]) - L.begin
    ();
    L[pos] = A[i];
    L_id[pos] = i;
    p[i] = pos ? L_id[pos-1] : -1;
    if (pos == k) {
    k = pos+1;
    }
  }
}
```

box_stacking.cpp

```
#include "../presets/base.cpp"

// We use the LIS idea of stacking values, finding the the largest
    height.
// As we can rotate the boxes and repeat them, we can get all
   permutations
// of height, lenght and depth and sort them to get the best order
    (a sequence
// that can maximize the stacking

int boxStacking(vector<vector<int>>& b_ext) {
  vector<vector<int>> boxes;
  for (auto box: b_ext) {
    boxes.push_back({box[0],box[1],box[2]});
    boxes.push_back({box[0],box[2],box[1]});
    boxes.push_back({box[1],box[0],box[2]});
    boxes.push_back({box[1],box[2],box[0]});
    boxes.push_back({box[2],box[0],box[1]});
    boxes.push_back({box[2],box[1],box[0]});
  }
  sort(boxes.rbegin(), boxes.rend());
  int n = boxes.size();
  // Use an array
  vector<int> dp(n, 0);
  int ans=0;

  for(int i=0;i<n;i++) {
    auto box = boxes[i];
    for(int j=0;j<i;j++) {
      if (box[0] < boxes[j][0] && box[1] < boxes[j][1]) {
        dp[i] = max(dp[j], dp[i]);
      }
    }
    dp[i] += box[2];
    ans = max(ans, dp[i]);
  }

  return ans;
}
```

edit_distance.cpp

```
#include "../presets/base.cpp"

int editDistance(string a, string b) {
  if (a.size() > b.size()) swap(a,b);
  int n = a.size(), m =b.size();

  vector<vector<int>> dp_(n+1, vector<int>(m+1, 0));
  for(int i=0;i<=n;i++) {
    dp_[i][0] = i;
  }
  for(int i=0;i<=m;i++) {
    dp_[0][i] = i;
  }

  for(int i=1;i<=n;i++) {
    for(int j=1;j<=m;j++) {
      if (a[i-1] == b[j-1]) {
        dp_[i][j] = dp_[i-1][j-1];
      } else {
        dp_[i][j] = min({dp_[i-1][j], dp_[i][j-1], dp_[i-1][j-1]})
     + 1;
      }
    }
  }

  return dp_[n][m];
}
```

subset_sum.cpp

```
vector<vector<bool>> dp(n+1, vector<bool>(max_sum+1, false));
dp[0][0] = true;
for(int j = 0; j<=max_sum; j++){
  for(int i = 1; i<=n; i++){
    dp[i][j] = dp[i-1][j] || (j >= w[i-1] ? dp[i-1][j-w[i-1]] : 0)
    ;
  }
}
```

tsp.cpp

```
#include "../presets/base.cpp"

const int MAXN = 20;
int dp[MAXN][1<<MAXN], cost[MAXN][MAXN];

// Is necessary change too many things during the contest
// but, its a good base for the problem
int tsp (int n) {
  memset(dp, 0x3f, sizeof(dp));

  for(int i=0;i<n-1;i++) {
    dp[i][1 << i] = cost[0][i+1];
  }

  for(int i=1;i<1<<(n-1);i++){
    for(int j=0;j<n-1;j++) {
      if (!(i & (1 << j))) continue; // not visited yet
      int value = dp[j][i];
      int u = j + 1;
      for(int k =0;k<n-1;k++) {
        if ((i & (1 << k))) continue; // already visited
        int newmask = i|(1 << k);
        dp[k][newmask] = min(dp[k][newmask], value + cost[u][k+1])
    ;
        }
    }
  }
  // INF
  int ans = INF;
  for(int i = 0;i<(n-1);i++){
    ans = min(ans, dp[i][(1<<(n-1))-1] + cost[i+1][0]);
  }

  return ans;
}
```

2 geometry

convex_hull.cpp

```
#include "geometry.cpp"
// From geometry using: Point, point::orientation,
struct Convex_hull{
  static bool cw(const point&a, const point&b, const point&c, bool
     include_collinear){
    // Easy to change to ccw, just change all the calls of cw to
    ccw
    int o = orientation(a-c, b-c);
    return o < 0 || (include_collinear && o == 0);
  }
  friend void graham_scan(vector<point> &a, bool include_collinear
     = false){
    point p0 = *min_element(a.begin(), a.end(), point::smallest_y)
    ;
    point::translat = p0;
    sort(a.begin(), a.end(), point::cw_cmp);
    // Often it will be requested to sort ccw, so just change the
    algorithm for that and it already works.
    if(include_collinear){
      int i = (int)a.size()-1;
      while(i >= 0 && collinear(p0, a[i], a.back())) i--;
      reverse(a.begin()+i+1, a.end());
    }
    vector<point> st;
    for(int i = 0; i<(int)a.size(); i++){
      while(st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i],
     include_collinear))
        st.pop_back();
      st.push_back(a[i]);
    }
    if(include_collinear == false && st.size() == 2 && st[0] == st
    [1])
      st.pop_back();
    a = st;
  }
};
```

convex_hull_trick.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
```

```
// Essa bizarrisse funcionou melhor do que a outra bizarrisse do
    cp algorithms então vamos com ela
// Teoricamente é pior na inserção de linhas, pois é n log n
    enquanto a outra é linear
// Porém funciona para mais casos pois a manutenção é dinâmica, nã
    o exige pré ordenamento

struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

geometry.cpp

```
#include "../presets/base.cpp"
const ld eps = 1e-9, inf = 1e9;

struct point {
  ld x, y;
  point(ld x = 0, ld y = 0) : x(x), y(y) {}
  static point origin;
  static point translat;

  friend point operator+(const point p, const point q) {
    return point(p.x + q.x, p.y + q.y);
  }
  friend point operator-(const point p, const point q) {
    return point(p.x - q.x, p.y - q.y);
  }
  friend point operator*(const point p, const ld k) {
    return point(p.x * k, p.y * k);
  }
  friend ld dot(const point p, const point q) {
    return p.x * q.x + p.y * q.y;
  }
  friend ld cross(const point p, const point q) {
    return p.x * q.y - p.y * q.x;
  }
  friend ld dist(const point &p, const point &q) {
    return sqrt(fabs(dot(p - q, p - q)));
  }
  friend ld proj(const point &p, const point &q) {
    return dot(p, q) / (dist(p, q));
  }
  bool operator<(const point &p) const { // Return smallest (x, y)
    return x < p.x || (x == p.x && y < p.y);
  }
  static bool smallest_y(const point&p, const point&q){
    return p.y < q.y || (p.y == q.y && p.x < q.x);
  }
  static bool angle_cmp(const point &p, const point &q) {
    auto op = p - translat;
    auto oq = q - translat;
    ld a = atan2l(op.y, op.x), b = atan2l(oq.y, oq.x);
    return a < b;
  }
  friend int orientation(const point &p, const point &q) {
    ld o = cross(p, q);
    if (o < 0) return -1; // clockwise
    if (o > 0) return 1;  // counter clockwise
    return 0;       // collinear
```

```
  }
  static bool cw_cmp(const point&p, const point &q){
    point v1 = p-translat;
    point v2 = q-translat;
    int o = orientation(v1, v2);
    if(o == 0){
      return dot(v1, v1) < dot(v2, v2);
    }
    return o < 0;
  }
  bool operator==(const point &p) {
    return (x == p.x && y == p.y);
  }
  friend bool collinear(point &a, point &b, point &c) {
    return orientation(a - c, b - c) == 0;
  }
  friend ostream &operator<<(ostream &os, const point &p) {
    os << p.x << ", " << p.y;
    return os;
  }
};
point point::origin(0, 0);
point point::translat(0, 0);

struct point3d {
  ld x, y, z;
  point3d() {}
  point3d(ld x, ld y, ld z) : x(x), y(y), z(z) {}
  friend point3d operator+(const point3d &p, const point3d &q) {
    return point3d(p.x + q.x, p.y + q.y, p.z + q.z);
  }
  friend point3d operator-(const point3d &p, const point3d &q) {
    return point3d(p.x - q.x, p.y - q.y, p.z - q.z);
  }
  friend point3d operator*(const point3d &p, ld q) {
    return point3d(p.x * q, p.y * q, p.z * q);
  }
  friend point3d operator/(const point3d &p, ld q) {
    return point3d(p.x / q, p.y / q, p.z / q);
  }
  friend ld dot(const point3d &p, const point3d &q) {
    return p.x * q.x + p.y * q.y + p.z * q.z;
  }
  friend point3d cross(const point3d &p, const point3d &q) {
    return point3d(
      p.y * q.z - p.z * q.y,
      p.z * q.x - p.x * q.z,
      p.x * q.y - p.y * q.x
    );
  }
};

struct halfplane {
  point p, pq;
  ld angle;
  halfplane() {}
  halfplane(point a, point b) : p(a), pq(b - a) {
    angle = atan2l(pq.y, pq.x);
  }
  bool out(const point &r) {
    return cross(pq, r - p) < -eps;
  }
  bool operator<(const halfplane &e) const {
    return angle < e.angle;
  }
  friend point intersection(const halfplane &s, const halfplane &t
    ) {
    ld alpha = cross((t.p - s.p), t.pq) / cross(s.pq, t.pq);
    return s.p + (s.pq * alpha);
  }
};

double area(const vector<point> &v) {
  double res = 0;
  for (int i = 0; i < v.size(); i++) {
    point p = i ? v[i - 1] : v.back();
    point q = v[i];
    res += (p.x - q.x) * (p.y + q.y);
  }
  return fabs(res) / 2;
}

int sgn(ld val) {
  return val > 0 ? 1 : (val == 0 ? 0 : -1);
}

point mass_center(const vector<point> &v) {
  ld x = 0, y = 0;
```

```
  int n = v.size();
  for (int i = 0; i < n; i++) {
    x += v[i].x;
    y += v[i].y;
  }
  return point(x / (ld)n, y / (ld)n);
}

bool pointInTriangle(point a, point b, point c, point p) {
  ld s1 = abs(cross(b - a, c - a));
  ld s2 = abs(cross(a - p, b - p)) + abs(cross(b - p, c - p)) +
    abs(cross(c - p, a - p));
  return s1 == s2;
}
```

half-plane_Intersection.cpp

```
#include "geometry.cpp"
#define hp halfplane
vector<point> hp_intersect(vector<hp> &h){
  point box[4] = {
    point(inf, inf),
    point(-inf, inf),
    point(-inf, -inf),
    point(inf, -inf)
  };
  for(int i = 0; i<4; i++){
    hp aux(box[i], box[(i+1)%4]);
    h.push_back(aux);
  }
  sort(h.begin(), h.end());
  deque<hp> dq;
  int len = 0;
  for(int i = 0; i < h.size(); i++){
    while(len > 1 && h[i].out(intersection(dq[len-1], dq[len-2])))
    {
      dq.pop_back();
      --len;
    }
    while(len > 1 && h[i].out(intersection(dq[0], dq[1]))){
      dq.pop_front();
      --len;
    }
    if(len > 0 && fabsl(cross(h[i].pq, dq[len-1].pq)) < eps){
      if(dot(h[i].pq, dq[len-1].pq) < 0.0)
        return vector<point>();
      if (h[i].out(dq[len-1].p)) {
        dq.pop_back();
        --len;
      }
      else continue;
    }

    dq.push_back(h[i]);
    ++len;
  }
  while (len > 2 && dq[0].out(intersection(dq[len-1], dq[len-2])))
    {
    dq.pop_back();
    --len;
  }
  while (len > 2 && dq[len-1].out(intersection(dq[0], dq[1]))) {
    dq.pop_front();
    --len;
  }
  if (len < 3) return vector<point>();

  vector<point> ret(len);
  for(int i = 0; i+1 < len; i++) {
    ret[i] = intersection(dq[i], dq[i+1]);
  }
  ret.back() = intersection(dq[len-1], dq[0]);
  return ret;
}
```

lichaotree.cpp

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
```

```
  return (conj(a) * b).x();
}

ftype f(point a,  ftype x) {
  return dot(a, {x, 1});
}

const int maxn = 2e5;

point line[4 * maxn];

void add_line(point nw, int v = 1, int l = 0, int r = maxn) {
  int m = (l + r) / 2;
  bool lef = f(nw, l) < f(line[v], l);
  bool mid = f(nw, m) < f(line[v], m);
  if(mid)
    swap(line[v], nw);
  if(r - l == 1)
    return;
  else if(lef != mid)
    add_line(nw, 2 * v, l, m);
  else
    add_line(nw, 2 * v + 1, m, r);
}

ftype get(int x, int v = 1, int l = 0, int r = maxn) {
  int m = (l + r) / 2;
  if(r - l == 1)
    return f(line[v], x);
  else if(x < m)
    return min(f(line[v], x), get(x, 2 * v, l, m));
  else
    return min(f(line[v], x), get(x, 2 * v + 1, m, r));
}
```

minkowski.cpp

```
// AKA ConvexHull sum, the set of points A, B, and C where C = {a+
    b | a e A, b e B}
#include "geometry.cpp"

void reorder_polygon(vector<point> & p){
  int pos = 0;
  for(int i = 1; i < p.size(); i++){
    if(p[i].y < p[pos].y || (p[i].y == p[pos].y && p[i].x < p[pos
    ].x))
      pos = i;
  }
  rotate(p.begin(), p.begin() + pos, p.end());
}

vector<point> minkowski(vector<point> p, vector<point> q){
  reorder_polygon(p);
  reorder_polygon(q);
  p.push_back(p[0]);
  p.push_back(p[1]);
  q.push_back(q[0]);
  q.push_back(q[1]);
  vector<point> result;
  int i = 0, j = 0;
  while(i < p.size() - 2 || j < q.size() - 2){
    result.push_back(p[i] + q[j]);
    auto c = cross(p[i+1] - p[i], q[j+1] - q[j]);
    if(c >= 0 && i < p.size() - 2)
      ++i;
    if(c <= 0 && j < q.size() - 2)
      ++j;
  }
  return result;
}
```

pointInConvexPolygon.cpp

```
#include "geometry.cpp"
vector<point> seq;
int n; point translation;
void prepare(vector<point> &v){ // just garantee that the sorted
    points begin at the lowest (x, y)
  int pos = 0;
  n = v.size();
  for (int i = 1; i < n; i++) {
    if (v[i] < v[pos]) pos = i;
  }
  rotate(v.begin(), v.begin() + pos, v.end());
  n--;
  seq.resize(n);
```

```
  for (int i = 0; i < n; i++)
    seq[i] = v[i + 1] - v[0];
  translation = v[0];
}

bool pointInConvexPolygon(point p){
  // must be a sorted convex polygon.
  p = p - translation;
  if (cross(seq[0], p) != 0 &&
      sgn(cross(seq[0], p)) != sgn(cross(seq[0], seq[n-1])))
    return false;
  if (cross(seq[n-1], p) != 0 &&
      sgn(cross(seq[n-1], p)) != sgn(cross(seq[n-1], seq[0])))
    return false;
  if (cross(seq[0], p) == 0)
    return dot(seq[0], seq[0]) >= dot(p, p);

  int l = 0, r = n-1;
  while (r - l > 1) {
    int mid = (l + r) / 2;
    int pos = mid;
    if (cross(seq[pos], p) >= 0)
      l = mid;
    else
      r = mid;
  }
  int pos = l;
  return pointInTriangle(seq[pos], seq[pos + 1], point::origin, p)
    ;
}
```

3 graphs

MST.cpp

```
#include "../presets/base.cpp"

struct DSU {
  vector<int> parent, size;
  DSU(int n) {
    parent.resize(n);
    size.resize(n);
    for (int i = 0; i < n; i++) {
      parent[i] = i;
      size[i] = 1;
    }
  }

  int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
  }
  void join(int a, int b) {
    a = find(a);
    b = find(b);
    if (a == b) return;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
  }
};

bool cmp(pair<pair<int, int>, int> &a, pair<pair<int, int>, int> &
    b){
  return a.second < b.second;
}

int MST(int n, vector<pair<int, int>, int> &edges, vector<pair<
    pair<int, int>, int>> &tree){
  int m = edges.size();
  sort(edges.begin(), edges.end(), cmp);
  DSU dsu(n);

  int count = 1;
  double sum = 0;
  for(int e = 0; e<m && count < n; e++){
    int u = edges[e].first.first, v = edges[e].first.second;
    double w = edges[e].second;
    if(dsu.find(u) != dsu.find(v)){
      dsu.join(u, v);
      tree.push_back({{u, v}, w});
      count++;
      sum += w;
    }
  }
  return sum;
```

```
}
```

```
};
```

bellmanFord.cpp

```
#include "../presets/base.cpp"
struct Graph {
  vector<vector<pair<int, int>>> adj;
  vector<int> dist;

  void create(int numberOfVertices) {
    adj.clear();
    adj.assign(numberOfVertices, {});
  }

  void addEdge(int u, int v, int w){
    adj[u].push_back({w, v});
  }

  //1 - if has negative cycles; 0 - if not
  int bellmanFord(int s) {
    dist.clear();
    dist.assign(adj.size(), 1e9);

    dist[s] = 0;
    int cycle = 0;
    for (int k = 0; k <= adj.size(); k++) {
      for (int u = 0; u < adj.size(); u++) {
        for (pair<int, int> edge: adj[u]) {
          int v = edge.second;
          int w = edge.first;

          if (dist[u]+w<dist[v]) {
            dist[v] = dist[u]+w;
            if (k == adj.size()) {
              cycle = 1;
            }
          }
        }
      }
    }

    return cycle;
  }
```

centroides.cpp

```
#include "../presets/base.cpp"
struct Centroid{
  vector<int> subtree_size;
  vector<bool> is_removed;
  vector<vector<int>> adj;
  vector<vector<pair<int, int>>> ancestroids;
  Centroid(int v, vector<vector<int>> &g){
    subtree_size.assign(v, 0);
    is_removed.assign(v, false);
    ancestroids.assign(v, vector<pair<int, int>>());
    adj = g;
    build(0);
  }

  int get_subtree_size(int u, int parent = -1){
    subtree_size[u] = 1;
    for(int v: adj[u]){
      if(v == parent || is_removed[v]) continue;
      subtree_size[u] += get_subtree_size(v, u);
    }
    return subtree_size[u];
  }

  int get_centroid(int u, int tree_size, int parent = -1){
    for(int v: adj[u]){
      if(v == parent || is_removed[v]) continue;
      if(subtree_size[v] * 2 > tree_size){
        return get_centroid(v, tree_size, u);
      }
    }
    return u;
  }

  void build(int u){
    int subtree_size = get_subtree_size(u);
    int centroid = get_centroid(u, subtree_size);

    for(int v: adj[centroid]){
      if(!is_removed[v]){
```

```
        set_dists(v, centroid, centroid, 1);
      }
    }
    is_removed[centroid] = true;

    for(int v: adj[centroid]){
      if(!is_removed[v]){
        build(v);
      }
    }
  }

  void set_dists(int v, int centroid, int parent, int dist){
    ancestroids[v].push_back({centroid, dist});
    for(int w: adj[v]){
      if (w == parent || is_removed[w]) continue;
      set_dists(w, centroid, v, dist+1);
    }
  }
};
```

dsu.cpp

```
#include "../presets/base.cpp"
struct DSU {
  vector<int> parent, size;
  void create(int n) {
    parent.resize(n);
    size.resize(n);
    for (int i = 0; i < n; i++) {
      parent[i] = i;
      size[i] = 1;
    }
  }

  int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
  }
  void join(int a, int b) {
    a = find(a);
    b = find(b);
    if (a == b) return;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
  }
};
```

dyjkstra.cpp

```
#include "../presets/base.cpp"
struct Graph {
  vector<vector<pair<int, int>>> adj;
  vector<int> dist;

  void create(int numberOfVertices) {
    adj.clear();
    adj.assign(numberOfVertices, {});
  }

  void addEdge(int u, int v, int w){
    adj[u].push_back({w, v});
  }

  void dyjkstra(int s) {
    dist.clear();
    dist.assign(adj.size(), 1e9);

    priority_queue<pair<int, int>> q;
    q.push({0, s});

    while (!q.empty()) {
      int u = q.top().second;
      int cost = q.top().first; q.pop();

      if (cost == dist[u]) {
        for (pair<int, int> edge: adj[u]) {
          int v = edge.second;
          int w = edge.first;

          if (cost+w < dist[v]) {
            dist[v] = cost+w;
            q.push({dist[v], v});
          }
        }
```

```
      }
    }
  }
};
```

hungarian.cpp

```
// Algoritmo do cp algorithms, usa o Kuhn porém não como uma função externa, pois é modificado e reutilizado
#include "../presets/base.cpp"
int hungarian(vector<vector<int>> &A){
  int n = A.size()-1, m = A[0].size()-1; // For some reason, the cp algorithm uses 1-index
  vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
  for (int i=1; i<=n; ++i) {
    p[0] = i;
    int j0 = 0;
    vector<int> minv (m+1, INF);
    vector<bool> used (m+1, false);
    do {
      used[j0] = true;
      int i0 = p[j0],  delta = INF,  j1;
      for (int j=1; j<=m; ++j)
        if (!used[j]) {
          int cur = A[i0][j]-u[i0]-v[j];
          if (cur < minv[j])
            minv[j] = cur,  way[j] = j0;
          if (minv[j] < delta)
            delta = minv[j],  j1 = j;
        }
      for (int j=0; j<=m; ++j)
        if (used[j])
          u[p[j]] += delta,  v[j] -= delta;
        else
          minv[j] -= delta;
      j0 = j1;
    } while (p[j0] != 0);
    do {
      int j1 = way[j0];
      p[j0] = p[j1];
      j0 = j1;
    } while (j0);
  }
  return -v[0];
}
```

kosaraju.cpp

```
#include "../presets/base.cpp"

void dfs(int u, vector<bool> &v, vector<vector<int>> &adj, vector<int> &out){
  v[u] = true;
  for(int e: adj[u]){
    if(!v[e]) dfs(e, v, adj, out);
  }
  out.push_back(u);
}

void SCC(vector<vector<int>> &adj, vector<vector<int>> &componentes){
  int n = adj.size();
  vector<vector<int>> adj_T(n, vector<int>());
  for(int i = 0; i<n; i++){
    for(int u: adj[i]){
      adj_T[u].push_back(i);
    }
  }
  vector<bool> visitados(n, false);
  vector<int> st;
  for(int i = 0; i<n; i++){
    if(visitados[i]) continue;
    dfs(i, visitados, adj_T, st);
  }
  visitados.assign(n, false);
  reverse(st.begin(), st.end());

  for(auto v: st){
    if(!visitados[v]){
      vector<int> component;
      dfs(v, visitados, adj, component);
      componentes.push_back(component);
    }
  }
}
```

kuhn.cpp

```
#include "../presets/base.cpp"
struct Kuhn{

  // Para grafos bipartidos
  int n, k; // n = primeira partição, k = segunda partição
  vector<vector<int>> g; // Lista de adjecências da primeira parti
    ção para a segunda.
  // Escolher a menor delas como a primeira durante a leitura.
  vector<int> mt;
  vector<bool> used;

  bool try_kuhn(int v) {
    if (used[v])
    return false;
    used[v] = true;
    for (int to : g[v]) {
      if (mt[to] == -1 || try_kuhn(mt[to])) {
        mt[to] = v;
        return true;
      }
    }
    return false;
  }

  void kuhn(){
    mt.assign(k, -1);
    vector<bool> used1(n, false);
    for (int v = 0; v < n; ++v) {
      for (int to : g[v]) {
        if (mt[to] == -1) {
          mt[to] = v;
          used1[v] = true;
          break;
        }
      }
    }
    for (int v = 0; v < n; ++v) {
      if (used1[v])
        continue;
      used.assign(n, false);
      try_kuhn(v);
    }
    for (int i = 0; i < k; ++i)
      if (mt[i] != -1)
        // mt[i] tells what vertex of the first part is connected
    to the vertex i of the second part, or -1 if it's not
    connected.
        // So if you want the size of it, just count how many aren
    't -1.
        printf("%d %d\n", mt[i] + 1, i + 1);
  }
};
```

lca.cpp

```
#include "../presets/base.cpp"
// LCA by binary_lifting

vector<int> st, en, depth; // depth is not needed for lca but for
    the virtual_tree
vector<vector<int>> adj, up; // up is the "ancestors" vector but
    up[i][j] is the 2^j ancestor of i.
int n, tmp, max_it;
void dfs(int v, int p){
  st[v] = ++tmp;
  up[v][0] = p;
  for(int i = 1; i<= max_it; i++){
    up[v][i] = up[up[v][i-1]][i-1];
  }
  for(int e: adj[v]){
    if(e != p){
      depth[e] = depth[v]+1;
      dfs(e, v);
    }
  }
  en[v] = ++tmp;
}
bool is_ancestor(int u, int v){
  return st[u] <= st[v] && en[v] <= en[u]; // v is above u
}

int lca(int u, int v){
  if(is_ancestor(u, v)) return u;
  if(is_ancestor(v, u)) return v;
```

```
  for(int i = max_it; i>=0; --i){
    if(!is_ancestor(up[u][i], v)) u = up[u][i];
  }
  return up[u][0];
}

void pre_compute_lca(int root){
  st.resize(n); en.resize(n); depth.resize(n);
  depth[root] = 0;
  tmp = 0;
  max_it = ceil(log2(n));
  up.assign(n, vector<int>(max_it+1));
  dfs(root, root);
}
```

topological_sort.cpp

```
#include "../presets/base.cpp"

void dfs(int u, vector<bool> &v, vector<vector<int>> &adj, vector<
    int> &out){
  v[u] = true;
  for(int e: adj[u]){
    if(!v[e]) dfs(e, v, adj, out);
  }
  out.push_back(u);
}

void topological_sort(vector<vector<int>> &adj, vector<int> &st){
  int n = adj.size();
  vector<vector<int>> adj_aux(n);
  for(int i = 0; i<n; i++){
    for(int u: adj[i])
      adj_aux[u].push_back(i);
  }
  vector<bool> visitados(n, false);

  for(int i = 0; i<n; i++){
    if(!visitados[i]){
      dfs(i, visitados, adj_aux, st);
    }
  }
}
```

virtual_tree.cpp

```
#include "lca.cpp"
// Solve problems like the query of a sum of the distance between
    all pair of given vertices in a tree
// It basically compress the tree to be just the important nodes (
    The considered vertices and their common ancestors),
// which are at most 2k-1, where k is the number of vertices
    considered (In k log k time).
struct Virtual_tree{
  vector<vector<int>> adj_vt;
  int vt_root;
  bool cmp(int u, int v){
    return st[u] < st[v];
  }
  Virtual_tree(int n, vector<int> &vert){
    adj_vt.assign(n, vector<int>());
    // Pick the needed vertices
    sort(vert.begin(), vert.end(), cmp);
    int k = vert.size();
    for(int i = 0; i < k-1; i++){
      vert.push_back(lca(vert[i], vert[i+1]));
    }
    sort(vert.begin(), vert.end(), cmp);
    vert.erase(unique(vert.begin(), vert.end()), vert.end()); //
    Erase duplicates

    // build the actually virtual tree
    vector<int> st; st.push_back(vert[0]);

    for(int i = 1; i<vert.size(); i++){
      int u = vert[i];
      while(st.size() >= 2 && !is_ancestor(st.back(), u)){
        // add edge to the tree
        adj_vt[st[st.size()-2]].push_back(st.back());
        // here only the top -> bottom is added, which is fine as
    we only need to transverse it from the root to the leaves.
        st.pop_back();
      }
      st.push_back(u);
    }
```

```
    while(st.size() >= 2){
      adj_vt[st[st.size()-2]].push_back(st.back());
      st.pop_back();
    }
    vt_root = st[0];
  }
};
```

4 math

divisors.cpp

```
#include "../presets/base.cpp"
// Get all divisors via a map with the prime factors
// Probably could be done better.
void divisors(map<int, int> &factors, set<int> &d){
  d.clear(); d.insert(1);
  for(auto e: factors){
    int p = e.first;
    int exp = e.second;
    vector<int> aux;
    for(auto c : d){
      long long temp = 1;
      for(int i = 0; i < exp; i++){
        temp *= p;
        aux.push_back(c * temp);
      }
    }
    for(auto v : aux) d.insert(v);
  }
}
```

ext_euclidean.cpp

```
#include "../presets/base.cpp"
// {gcd, x, y}
tuple<int, int, int> gcd(int a, int b) {
  if(b == 0) return make_tuple(a, 1, 0);
  int q, w, e;
  tie(q, w, e) = gcd(b, a % b);
  return make_tuple(q, e, w - e * (a / b));
}
```

factorize.cpp

```
#include "../presets/base.cpp"
// Faster option: while(a % 2) and while(a % 3) first
// and after that i = 5, i+=6, (a % i) and (a % (i+2))
void factorize(int a, map<int, int> &factors){
  for(int i = 2; i*i<=a; i++){
    while(a%i == 0){
      factors[i]++;
      a/=i;
    }
  }
  if(a > 1) factors[a]++;
}
```

fast_pow.cpp

```
#include "../presets/base.cpp"
#define md 1000000007
int fast_pow(int a, int b){ // Same logic (and almost the same
    code) can be applied
              // to all comutative and associative operations, sum
    , gcd, etc...
   int res = 1;
   while(b){
     if(b & 1){
       res = (a*b)%md;
     }
     a = (a*a)%md;
     b >>=1;
   }
   return res;
 }
```

5 presets

base.cpp

```
#include<bits/stdc++.h>
#define ll long long
#define ld long double
#define INF INT_MAX

using namespace std;

void fast_io(){
  cin.tie(0);
  cout.tie(0);
  ios_base::sync_with_stdio(0);
}

int main(){

}
```

gen.cpp

```
#include <bits/stdc++.h>
using namespace std;

int rand(int a, int b){
  return a + rand()%(b - a + 1);
}
// Generate a random input for the sh script. Should be modified
    based on the real problem.
int main(int argc, char *argv[]){
  srand(atoi(argv[1]));
  int n = rand(2, 5000);
  printf("%d\n", n);
  set<int> used;
  for(int i = 0; i<n; i++){
    int x;
    do {
      x = rand(1, 5000);
    } while(used.count(x));
    printf("%d ", x);
    used.insert(x);
  }
  puts("");
}
```

6 ranges

bit.cpp

```
template<int amountOfPicks>
class BIT{
 public:
  int range[amountOfPicks+1];
  int read(int index) {
    index++;
    int runningSum = 0;
    while (index > 0) {
      runningSum += range[index];
      int rightMostSetBit = (index & (-index));
      index -= rightMostSetBit;
    }

    return runningSum;
  }

  int readRange(int l, int r) {
    return read(r) - read(l);
  }

  void clear() {
    memset(range, 0, sizeof(int) * amountOfPicks);
  }

  void update(int index, int x) {
    index++;
    while (index < amountOfPicks) {
      range[index] += x;
      int rightMostSetBit = (index & (-index));
      index += rightMostSetBit;
    }
  }
```

```
  void updateRange(int l, int r, int x) {
    update(l, x);
    update(r+1, -x);
  }
};
```

segtree.cpp

```
#include "../presets/base.cpp"
// To change the operation, just change conquer and RSQ and
    propagate
class SegTree{
private:
  int n;
  vector<ll> A, St, Lazy;

  int rl(int p) {return p<<1;}
  int rr(int p) {return (p<<1)+1;}


  ll conquer(ll a, ll b) {
    return a+b;
  }

  void build(int i, int l, int r) {
    if(l == r) {
      St[i] = A[l];
    } else {
      int mid = (l+r)/2;
      build(rl(i), l, mid);
      build(rr(i), mid+1, r);
      St[i] = conquer(St[rl(i)], St[rr(i)]);
    }
  }

  void propagate(int i, int l, int r) {
    if (Lazy[i] != -1) {
      St[i] = Lazy[i] * (r - l + 1);
      if (l != r) { Lazy[rl(i)] = Lazy[rr(i)] = Lazy[i]; }
      else { A[l] = Lazy[i]; }
      Lazy[i] = -1;
    }
  }

  ll RSQ(int i, int l, int r, int tl, int tr) {
    propagate(i, tl, tr);
    if (l > tr || r < tl) return 0;
    if ((l <= tl) && (r >= tr)) return St[i];
    int mid = (tl+tr)/2;
    return conquer(RSQ(rl(i), l , r, tl, mid), RSQ(rr(i), l, r,
    min(tr,mid+1), tr));
  }

  void update(int i, int l, int r, int tl, int tr, int x) {
    propagate(i, l, r);
    if (l > tr || r < tl) return;
    if ((l <= tl) && (r >= tr)) {
      Lazy[i] = x;
      propagate(i, tl, tr);
    } else {
      int mid = (tl + tr) / 2;
      update(rl(i), l, r, tl, mid, x);
      update(rr(i), l, r, min(mid+1, tr), tr, x);
      St[i] = St[rl(i)] + St[rr(i)];
    }
  }

public:
  SegTree(int sz): n(sz), St(4*n), Lazy(4*n, -1) {}
  SegTree(const vector<ll> &initial):SegTree(initial.size()) {
    A = initial;
    build(1, 0, n-1);
  }

  void update(int i, int j, int val) { update(1,i,j, 0,n-1, val);}
  ll RSQ(int i, int j) { return RSQ(1, i,j,0,n-1);}
};
```

sparse_table.cpp

```
#include "../presets/base.cpp"
struct SparseTable {
  vector<vector<int>> st;
  vector<int> log;
  int n, k;
  SparseTable(vector<int> &a) {
```

```
    n = a.size();
    k = log2(n)+1;
    st.assign(k, vector<int>(n));
    copy(a.begin(), a.end(), st[0]);
    log.assign(n+1, 0);
    for(int i = 2; i<=n; i++) log[i] = log[i/2]+1;
    // It's possible to construct the st with any operation that
    satisfies f((a, b), c) = f(a, (b, c))
    // just switch the min for the desired function
    for (int i = 1; i <= k; i++)
      for (int j = 0; j + (1 << i) <= n; j++)
        st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))
    ]);
  }
  int query(int l, int r) {
    int i = log2(r - l + 1);
    return min(st[i][l], st[i][r - (1 << i) + 1]);
  }
  int sum_query(int l, int r){
    int sum = 0;
    for (int i = k; i >= 0; i--) {
      if ((1 << i) <= r - l + 1) {
        sum += st[i][l];
        l += 1 << i;
      }
    }
    return sum;
  }
};
```

7 strings

kmp.cpp

```
#include "../presets/base.cpp"
#include "prefix_function.cpp"
vector<int> kmp(string &t, string &p){
  vector<int> lps = prefix_function(t);
  int i = 0, j = 0;
  vector<int> res;
  while (i < t.size()) {
    if (t[i] == p[j]) {
      i++, j++;
      if (j == p.size()) {
        res.push_back(i-j);
        j = lps[j-1];
      }
    } else {
      if (j != 0) {
        j = lps[j-1];
      } else {
        i++;
      }
    }
  }
  return res;
}
```

prefix_function.cpp

```
#include "../presets/base.cpp"
vector<int> prefix_function(string &s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i-1];
    while (j > 0 && s[i] != s[j])
      j = pi[j-1];
    if (s[i] == s[j])
      j++;
    pi[i] = j;
  }
  return pi;
}
```